TURBO C®

VERSION 1.5

ADDITIONS & ENHANCEMENTS

BORLAND

TURBO C®

Addendum

Version 1.5
Additions and Enhancements

Borland International
4585 Scotts Valley Drive
Scotts Valley, CA 95066

This manual was produced in its entirety with
Sprint:® The Professional Word Processor,
available from Borland.

Borland International, Inc.
4585 Scotts Valley Drive
Scotts Valley, California 95066

10 9 8 7 6 5 4

Table of Contents

I N T R O D U C T I O N

Welcome to Turbo C version 1.5! This enhancement package includes a number of new Turbo C features. The major ones are

- more than 100 new functions, including powerful text and graphics video functions
- an object code librarian, so you can create and manage .LIB files
- new "creature comforts," such as 43- and 50-line support in the Integrated Environment and multiple library directories

In this addendum to the Turbo C manuals, we document the version 1.5 additions and enhancements. This addendum supplements your *Turbo C User's Guide* and *Turbo C Reference Guide*; refer to this addendum for information about new program features and any significant changes to the original manuals.

What You Will Find in this Addendum

This addendum has six chapters and four appendixes, covering the major differences between versions 1.0 and 1.5. Here's a summary of those differences, and the addendum chapters in which you will find information about them.

Turbo C's Video Functions (Chapter 1)

Turbo C's extended console I/O package (**cprintf**, **cputs**, etc.) provides powerful text-mode screen- and window-management capabilities, along with text-attribute control. The new BGI (Borland Graphics Interface) graphics library supplies versatile drawing/painting and graphics text-output functions. These graphics functions support CGA, EGA, Hercules, VGA, and other graphics adapters. If you are not familiar with video functions, windows, or graphics in general, read Chapter 1 in the addendum for a basic overview of these features and functions. Refer to Chapter 4 in the addendum for individual function descriptions.

New and revised menus, and new hot key (Chapter 2)

The original Options/Environment menu in Turbo C's integrated environment has been split into two menus (Environments and Directories) with some added options, and a new hot key provides additional functionality. The most notable feature on the new Options/Environments menu is the Screen size menu item—which lets you change the integrated environment display from 25-line mode to 43-line mode on your EGA-equipped system (or 50-line mode on your VGA-equipped system)—while the major feature on the new Options/Directories menu is the modified Library directories menu item.

The new capabilities on these menus include

- 25-, 43-, or 50-line display modes
- multiple library directories
- user-named pick files
- user-set tab sizes in the editor
- auto save of configuration file

The new hot key is *Alt-F5* (flip to/from saved output screen).

Multiple library directories (Chapters 2 and 3)

You can now give TCC multiple `-L`<*dirname*> options, just as you always could with the `-I` option. You can also list multiple library directories in the integrated environment, under the (new) Options/Directories menu.

Expanded command-line syntax (Chapter 3)

The syntax for the `-I` and `-L` command-line options has been expanded. These options now accept multiple directories, just as the integrated environment's equivalent menu items do.

New and modified functions and variables (Chapter 4)

Version 1.5 has over 100 new functions, including several powerful additions to the console (text) I/O functions, a whole library of video graphics functions, a handful of miscellaneous new functions for ANSI-compatibility, a few modifications to existing functions, plus some new (and some modified) global variables. These are all presented with complete descriptions that supplement chapters 1 and 2 of the *Turbo C Reference Guide.*

Revised function prototypes (Chapter 5)

To provide enhanced compatibility with the Proposed Draft ANSI C Standard, some of Turbo C's function prototypes have been revised in version 1.5. These prototypes are listed in an alphabetical table in this chapter.

Miscellaneous Information (Chapter 6)

This chapter documents miscellaneous changes and additions to the product and the manuals that don't fall under any of the preceding categories.

- The former CNVTCFG.EXE utility has been renamed TCCONFIG.EXE; you use it to convert back and forth between the configuration file for TCC (TURBOC.CFG) and those for TC (*.TC files).
- The search rules for MAKE's default file BUILTINS.MAK have changed; the new search algorithm is covered in this chapter.
- Version 1.5 supports two more predefined streams: *stdaux* and *stdprn*. All five predefined streams are explained in a section in this chapter.
- If you are not familiar with pick files and pick lists, refer to this chapter for a discussion.
- If you want to know more about configuration files, check out this chapter for an overview.
- Minor corrections to the original *Turbo C User's Guide* and *Turbo C Reference Guide* are listed here by page number.

New and revised utilities (Appendixes A, B, C, and D)

Version 1.5 includes one modified utility and three new ones, explained in these four appendixes.

- Appendix A covers TCINST.EXE, the optional custom installation program. One nice feature in the new TCINST is the ability to rebind editor command keystrokes (both secondary and primary) to your preferred key sequences.
- Appendix B describes TLIB.EXE, an object code librarian.
- Appendix C covers GREP.COM, a very fast version of the well-known Unix file-search utility.
- Appendix D explains how to use BGIOBJ.EXE, a graphics utility, when registering graphics drivers and character fonts in your programs.

C H A P T E R

1

Turbo C's Video Functions

In this chapter, we first briefly discuss video modes and windows. After those overviews, we describe programming in text mode, then in graphics mode.

Turbo C's new video functions are based on corresponding routines in Turbo Pascal 4.0. If you are not already familiar with controlling your PC's screen modes or creating and managing windows and viewports, take a few minutes to read the following words on those topics.

Some Words About Video Modes

Your PC has some kind of video adapter. This can be a Monochrome Display Adapter (MDA) for your basic text-only display, or it can be capable of displaying graphics, such as a Color Graphics Adapter (CGA), an Enhanced Graphics Adapter (EGA), or a Hercules Monochrome Graphics Adapter. Each adapter can operate in a variety of modes; the mode specifies whether the screen displays 80 or 40 columns (text mode only), the display resolution (graphics mode only), and the display type (color, monochrome, or black & white).

The screen's operating mode is defined when your program calls one of the mode-defining functions (**textmode**, **initgraph**, or **setgraphmode**).

- In *text mode*, your PC's screen is divided into cells (80 or 40 columns wide by 25 lines high). Each cell consists of an attribute and a character. The character is the displayed ASCII character, while the attribute specifies *how* the character is displayed (its color, intensity, etc.). Turbo C version

1.5 provides a full range of routines to manipulate the text screen: for writing text directly to the screen, and for controlling the cell attributes.

- In *graphics mode,* your PC's screen is divided into pixels; each pixel displays a single dot on the screen. The number of pixels (the resolution) depends on the type of video adapter connected to your system and the mode that adapter is in. You can use functions from Turbo C's new graphics library to create graphic displays on the screen: you can draw lines and shapes, fill enclosed areas with patterns, and control the color of each pixel.

In text modes, the upper-left hand corner of the screen is position (1,1), with *x*-coordinates increasing from left-to-right, and *y*-coordinates increasing from screen-top to screen-bottom. In graphics modes, the upper-left hand corner is position (0,0), with the *x*- and *y*-coordinate values increasing in the same manner.

Some Words About Windows and Viewports

Version 1.5 of Turbo C provides functions for creating and managing windows on your screen in text mode (and viewports in graphics mode). If you are not familiar with windows and viewports, you should read this brief overview. Turbo C's new window- and viewport-management functions are explained in "Programming in Text Mode" and "Programming in Graphics Mode" later in this chapter.

What is a Window?

A window is a rectangular area defined on your PC's video screen when it's in a text mode. When your program writes to the screen, its output is restricted to the active window. The rest of the screen (outside the window) remains untouched.

The default window is a full-screen text window. Your program can change this default full-screen text window to a text window smaller than the full screen (with a call to the **window** function). This function specifies the window's position in terms of screen coordinates.

What is a Viewport?

In graphics mode, you can also define a rectangular area on your PC's video screen; this is a viewport. When your graphics program outputs

drawings, etc., the viewport acts as the virtual screen. The rest of the screen (outside the viewport) remains untouched. You define a viewport in terms of screen coordinates with a call to the **setviewport** function.

Coordinates

Except for these window- and viewport-defining functions, all coordinates for text-mode and graphics-mode functions are given in window- or viewport-relative terms, not in absolute screen coordinates. The upper left corner of the text-mode window is the coordinate origin, referred to as (1,1); in graphics modes, the viewport coordinate origin is position (0,0).

Programming in Text Modes

In this section we give a brief summary of the functions you use in text mode: For more detailed information about these functions, refer to the function lookup section of this addendum.

In version 1.5 of Turbo C, the direct console I/O package (**cprintf**, **cputs**, etc.) has been enhanced to provide higher-performance text output, and extended to provide window management, cursor positioning, and attribute control functions. These functions are all part of the standard Turbo C libraries; they are prototyped in the header file CONIO.H.

The Console I/O Functions

Turbo C's text-mode functions work in any of the five possible video text modes: Which modes are available on your system depends on the type of video adapter and monitor you have. You specify the current text mode with a call to **textmode**. How to use this function is described later in this chapter, and under the **textmode** entry in Chapter 4 of this addendum.

These text-mode functions are divided into four separate groups:

- text output and manipulation
- window and mode control
- attribute control
- state query

We cover these four text-mode function groups in the following sections.

Text Output and Manipulation

Here's a quick summary of the text output and manipulation functions:

Writing and reading text:

cprintf	sends formatted output to the screen
cputs	sends a string to the screen
putch	sends a single character to the screen
getche	reads a character and echoes it to the screen

Manipulating text (and the cursor) on-screen:

clrscr	clears the text window
clreol	clears from the cursor to the end of the line
delline	deletes the line where the cursor rests
gotoxy	positions the cursor
insline	inserts a blank line below the line where the cursor rests
movetext	copies text from one area on screen to another

Moving blocks of text into and out of memory:

gettext	copies text from an area on screen to memory
puttext	copies text from memory to an area on screen

Your screen-output programs will come up in a full-screen text window by default, so you can immediately write, read, and manipulate text without any preliminary mode-setting. You write text to the screen with the direct console output functions **cprintf**, **cputs**, and **putch**, and echo input with the function **getche**. Text wraps within the window just as expected; if a word would extend beyond the window's right border, it is moved down to the beginning of the next line.

Once your text is on the screen, you can erase the active window with **clrscr**, erase part of a line with **clreol**, delete a whole line with **delline**, and insert a blank line with **insline**. The latter three functions operate relative to the cursor position; you move the cursor to a specified location with **gotoxy**. You can also copy a whole block of text from one rectangular location in the window to another with **movetext**.

You can capture a rectangle of on-screen text to memory with **gettext**, and put that text back on the screen (anywhere you want) with **puttext**.

Window and Mode Control

There are two window- and mode-control functions:

===

textmode	sets the screen to a text mode
window	defines a text-mode window

===

You can set your screen to any of several video text modes with **textmode** (limited only by your system's type of monitor and adapter). This initializes the screen as a full-screen text window, in the particular mode specified, and clears any residual images or text.

When your screen is in a text mode, you can output to the full screen, or you can set aside a *portion* of the screen—a window—to which your program's output is confined. To create a text window, you call **window**, specifying what area on the screen it will occupy.

Attribute Control

Here's a quick summary of the text-mode attribute control functions:

===

Setting foreground and background:

textcolor	sets the foreground color (attribute)
textbackground	sets the background color (attribute)
textattr	sets the foreground and background colors (attributes) at the same time

Converting intensity:

highvideo	sets text to high intensity
lowvideo	sets text to low intensity
normvideo	sets text to original intensity

===

The attribute-control functions set the current attribute, which is represented by an 8-bit value: the four lowest bits represent the foreground color, the next three bits give the background color, and the high bit is the "blink enable" bit.

Subsequent text is displayed in the current attribute. With the attribute-control functions, you can set the background and foreground (character) colors separately (with **textbackground** and **textcolor**) or combine the color specifications in a single call to **textattr**. You can also specify that the character—the foreground—will blink. Most color monitors in color modes will display the true colors. Non-color monitors may convert some or all of the attributes to various monochromatic shades or other visual effects, such as bold, underscore, reverse video, etc.

You can direct your system to map the high-intensity foreground colors to low intensity colors with **lowvideo** (which turns *off* the high intensity bit for the characters). Or you can map the low-intensity colors to high intensity with **highvideo** (which turns *on* the character high-intensity bit). When you're through playing around with the character intensities, you can restore the settings to their original values with **normvideo**.

State Query

Here's a quick summary of the state-query functions:

gettextinfo	fills in a **text_info** structure with information about the current text window
wherex	gives the *x*-coordinate of the cell containing the cursor
wherey	gives the *y*-coordinate of the cell containing the cursor

Turbo C's console I/O functions include some designed for "state query". With these functions, you can retrieve information about your text-mode window and the current cursor position within the window.

The **gettextinfo** function fills a **text_info** structure (defined in CONIO.H) with several details about the text window, including:

- the current video mode
- the window's position in absolute screen coordinates
- the window's dimensions
- the current foreground and background colors
- the cursor's current position

Sometimes you might need only a few of these details. Rather than retrieving all the text window information, you can find out just the cursor's (window-relative) position with **wherex** and **wherey**.

Text Windows

The default text window is full screen; you can change this to a less-than-full-screen text window with a call to the **window** function. Text windows can contain up to 25 lines (the maximum number of lines on-screen in any text mode) and up to 40 or 80 columns (depending on your text mode).

The coordinate origin of a Turbo C text window is the *upper left-hand corner* of the *window*. The coordinates of the window's upper left corner are (1,1); the coordinates of the bottom right corner of a full-screen 80-column text window are (80,25).

An Example

Suppose your 100% PC-compatible system is in 80-column text mode, and you want to create a window. The upper left corner of the window will be at screen coordinates (10, 8), and the lower right corner of the window will be at screen coordinates (50, 21). To do this, you call the **window** function, like this:

```
window(10, 8, 50, 21);
```

Now that you've created the text-mode window, you want to move the cursor to the *window* position (5, 8) and write some text in it, so you decide to use **gotoxy** and **cputs**.

```
gotoxy(5, 8);
cputs("Happy Birthday, Frank Borland");
```

Figure 1.1 illustrates these ideas.

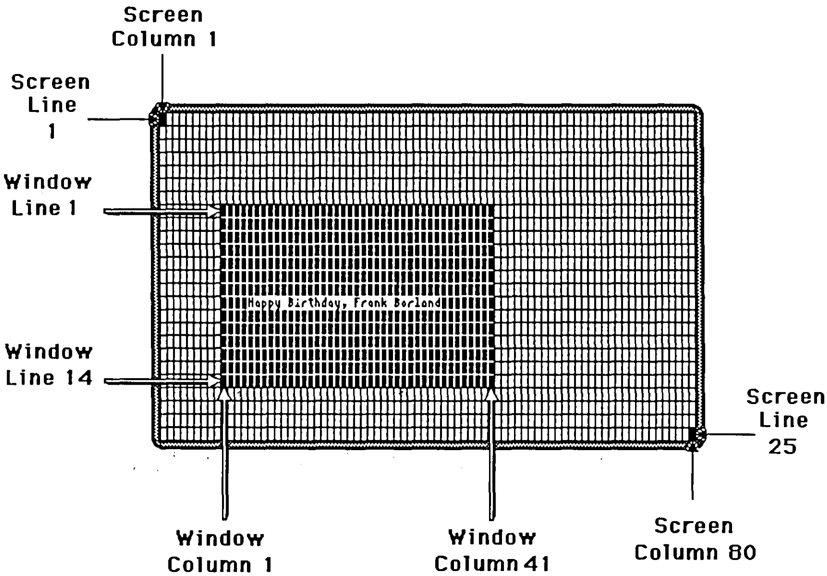

Figure 1.1: A Window in 80x25 Text Mode

The text_modes Type

You can put your monitor into one of five PC text modes with a call to the **textmode** function. The enumeration type *text_modes*, defined in CONIO.H, enables you to use symbolic names for the *mode* argument to the **textmode** function, instead of "raw" mode numbers. However, if you use the symbolic constants, you must `#include <conio.h>` in your source code.

The numeric and symbolic values defined by *text_modes* are as follows:

Symbolic Constant	*Numeric Value*	*Video Text Mode*
LAST	–1	Previous text mode enabled
BW40	0	Black & White, 40 columns
C40	1	16-Color, 40 columns
BW80	2	Black & White, 80 columns
C80	3	16-Color, 80 columns
MONO	7	Monochrome, 80 columns

For example, the following calls to **textmode** will put your color monitor in the indicated operating mode:

Call	*Operating Mode*
`textmode(0)`	Black&White, 40 column
`textmode(BW80)`	Black&White, 80 column
`textmode(C40)`	16-Color, 40 column
`textmode(3)`	16-Color, 80 column

Text Colors

For a detailed description of how cell attributes are laid out, refer to the **textattr** *entry in Chapter 4 of this addendum.*

When a character occupies a cell, the color of the character is the *foreground;* the color of the cell's remaining area is the *background.* Color monitors with color video adapters can display up to 16 different colors; monochrome monitors substitute different visual attributes (highlighted, underscored, reverse video, etc.) for the colors.

The include file CONIO.H defines symbolic names for the different colors. If you use the symbolic constants, you must `#include <conio.h>` in your source code.

The following table lists these symbolic constants and their corresponding numeric values. Note that only the first eight colors are available for the

background, while all sixteen colors are available for the foreground (the characters themselves).

Symbolic Constant	Numeric Value	Foreground or Background?
BLACK	0	both
BLUE	1	both
GREEN	2	both
CYAN	3	both
RED	4	both
MAGENTA	5	both
BROWN	6	both
LIGHTGRAY	7	both
DARKGRAY	8	foreground only
LIGHTBLUE	9	foreground only
LIGHTGREEN	10	foreground only
LIGHTCYAN	11	foreground only
LIGHTRED	12	foreground only
LIGHTMAGENTA	13	foreground only
YELLOW	14	foreground only
WHITE	15	foreground only
BLINK	128	foreground only

You can add the symbolic constant BLINK (numeric value 128) to a foreground argument if you want the character to blink.

High-Performance Output: the directvideo Variable

Turbo C's console I/O package includes a variable called *directvideo*. This variable controls whether your program's console output goes directly to the video RAM (*directvideo* = 1) or goes via BIOS calls (*directvideo* = 0).

The default value is *directvideo* = 1 (console output goes directly to the video RAM). In general, going directly to video RAM gives very high performance (spelled f-a-s-t-e-r o-u-t-p-u-t), but doing so requires your computer to be 100% IBM PC compatible: your video hardware must be identical to IBM display adapters. Setting *directvideo* = 0 will work on any machine that is IBM BIOS-compatible, but the console output will be slower.

Programming in Graphics Mode

In this section we give a brief summary of the functions you use in graphics mode: For more detailed information about these functions, refer to Chapter 4 of this addendum.

Turbo C version 1.5 provides a separate library of over 70 graphics functions, ranging from high-level calls (like **setviewport**, **bar3d**, and **drawpoly**) to bit-oriented functions (like **getimage** and **putimage**). The graphics library supports numerous fill and line styles, and provides several text fonts that you can magnify, justify, and orient horizontally or vertically.

These functions are in the new library GRAPHICS.LIB, and they are prototyped in the header file GRAPHICS.H. In addition to these two files, the graphics package includes graphics device drivers (*.BGI files) and stroked character fonts (*.CHR files); we discuss these additional files in following sections.

To use any of the graphics functions, you need to name GRAPHICS.LIB in your project file if you're using TC.EXE; if you're using TCC.EXE, you need to list GRAPHICS.LIB on the command line. For example, if your program, MYPROG.C, uses graphics, the TCC command line would be:

```
tcc myprog graphics.lib
```

For TC.EXE, your project file, MYPROG.PRJ, would contain the line

```
myprog graphics.lib
```

Important Note: There is only one graphics library, not separate versions for each memory model (in contrast to the standard libraries CS.LIB, CC.LIB, CM.LIB, etc., which are memory-model specific). Each function in GRAPHICS.LIB is a `far` function, and those graphics functions that take pointers take `far` pointers. For these functions to work properly, it is important that you `#include <graphics.h>` in every module that uses graphics.

The Graphics Library Functions

Turbo C's graphics functions comprise seven categories:

- graphics system control
- drawing and filling
- manipulating screens and viewports
- text output
- color control
- error handling
- state query

Graphics System Control

Here's a quick summary of the graphics system control functions:

closegraph	shuts down the graphics system
detectgraph	checks the hardware and determines which graphics driver and mode to use
graphdefaults	resets all graphics system variables to their default settings
_graphfreemem	deallocates graphics memory; hook for defining your own routine
_graphgetmem	allocates graphics memory; hook for defining your own routine
getgraphmode	returns the current graphics mode
getmoderange	returns lowest and highest valid modes for specified driver
initgraph	initializes the graphics system and puts the hardware into graphics mode
registerbgidriver	registers a linked-in or user-loaded driver file for inclusion at link time
restorecrtmode	restores the original (pre-**initgraph**) screen mode
setgraphbufsize	specifies size of the internal graphics buffer
setgraphmode	selects the specified graphics mode, clears the screen, and restores all defaults

Turbo C's graphics package provides graphics drivers for the following graphics adapters (and true compatibles):

- Color Graphics Adapter (CGA)
- Multi Color Graphics Array (MCGA)
- Enhanced Graphics Adapter (EGA)
- Video Graphics Array (VGA)
- Hercules Graphics Adapter
- AT&T 400-line Graphics Adapter
- 3270 PC Graphics Adapter

To start the graphics system, you first call the **initgraph** function. **initgraph** loads the graphics driver and puts the system into graphics mode. You can tell **initgraph** to use a particular graphics driver and mode, or to auto detect the attached video adapter at run time and pick the corresponding driver. If you tell **initgraph** to auto detect, it calls **detectgraph** to select a graphics driver and mode.

A graphics driver can support several different graphics modes. You find out how many modes a given driver supports with **getmoderange**, and what the current mode is with **getgraphmode**. You change graphics modes with **setgraphmode**, and can return the video mode to its original state (before graphics was initialized) with **restorecrtmode**.

graphdefaults resets the graphics state's settings (viewport size, draw color, fill color and pattern, etc.) to their default values.

Finally, when you're through using graphics, call **closegraph** to shut down the graphics system. **closegraph** unloads the driver from memory and restores the original video mode (via **restorecrtmode**).

A More Detailed Discussion

The previous discussion provided an overview of how **initgraph** *operates. In the following paragraphs, we describe the behavior of* **initgraph, _graphgetmem,** *and* **_graphfreemem** *in some detail.*

Normally, the **initgraph** routine loads a graphics driver by allocating memory for the driver, then loading the appropriate .BGI file from disk. As an alternative to this dynamic loading scheme, you can link a graphics driver file (or several of them) directly into your executable program file. You do this by first converting the .BGI file to an .OBJ file (using the BGIOBJ utility), then placing calls to **registerbgidriver** in your source code (before the call to **initgraph**) to *register* the graphics driver(s). When you build your program, you need to link the .OBJ files for the registered drivers.

After determining which graphics driver to use (perhaps *via* **detectgraph**), **initgraph** checks to see if the desired driver has been registered. If so, **initgraph** uses the registered driver directly from memory. Otherwise, **initgraph** allocates memory for the driver and loads the .BGI file from disk.

Note: Using **registerbgidriver** is an advanced programming technique, not recommended for novice programmers. This function is described in more detail in Appendix D in this addendum.

During run time, the graphics system might need to allocate memory for drivers, fonts, and internal buffers. If this is necessary, it calls **_graphgetmem** to allocate memory, and calls **_graphfreemem** to free it. By default, these routines simply call **malloc** and **free**, respectively.

You can override this default behavior by defining your own **_graphgetmem** and **_graphfreemem** functions. By doing this, you can control graphics memory allocation yourself. You must, however, use the same names for your own versions of these memory-allocation routines: they will override the default functions with the same names that are in the standard C libraries.

Drawing and Filling

Here's a quick summary of the drawing and filling functions:

==

Drawing:

arc	draws a circular arc
circle	draws a circle
drawpoly	draws the outline of a polygon
ellipse	draws an elliptical arc
getarccoords	returns the coordinates of the last call to **arc** or **ellipse**
getaspectratio	returns the aspect ratio of the current graphics mode
getlinesettings	returns the current line style, line pattern, and line thickness
line	draws a line from (*x0*, *y0*) to (*x1*, *y1*)
linerel	draws a line to a point some relative distance from the current position (CP)
lineto	draws a line from the current position (CP) to (*x*,*y*)
moveto	moves the CP to (*x*,*y*)
moverel	moves the current position (CP) a relative distance
rectangle	draws a rectangle
setlinestyle	sets the current line width and style

Filling:

bar	draws and fills a bar
bar3d	draws and fills a 3-D bar
fillpoly	draws and fills a polygon
floodfill	flood-fills a bounded region
getfillpattern	returns the user-defined fill pattern
getfillsettings	returns information about the current fill pattern and color
pieslice	draws and fills a pie slice
setfillpattern	selects a user-defined fill pattern
setfillstyle	sets the fill pattern and fill color

==

With Turbo C's drawing and painting functions, you can draw colored lines, arcs, circles, ellipses, rectangles, pieslices, 2- and 3-dimensional bars, polygons, and regular or irregular shapes based on combinations of these. You can fill any bounded shape (or any region surrounding such a shape) with one of 11 predefined patterns, or your own user-defined pattern. You can also control the thickness and style of the drawing line, and the location of the CP.

You draw lines and unfilled shapes with the functions **arc, circle, drawpoly, ellipse, line, linerel, lineto,** and **rectangle.** You can fill these shapes with

floodfill, or combine drawing/filling into one step with **bar**, **bar3d**, **fillpoly**, and **pieslice**. You use **setlinestyle** to specify whether the drawing line (and border line for filled shapes) is thick or thin, and whether its style is solid, dotted, etc., or some other line pattern you've defined. You can select a predefined fill pattern with **setfillstyle**, and define your own fill pattern with **setfillpattern**. You move the current position (CP) to a specified location with **moveto**, and move it a specified *distance* with **moverel**.

To find out the current line style and thickness, you call **getlinesettings**. For information about the current fill pattern and fill color, you call **getfillsettings**; you can get the user-defined fill pattern with **getfillpattern**.

You can get the aspect ratio (the scaling factor used by the graphics system to make sure circles come out round) with **getaspectratio**, and get coordinates of the last drawn arc or ellipse by calling **getarccoords**.

Manipulating the Screen and Viewport

Here's a quick summary of the image-manipulation functions:

Screen Manipulation

cleardevice	clears the screen
setactivepage	sets the active page for graphics output
setvisualpage	sets the visual graphics page number

Viewport Manipulation

clearviewport	clears the current viewport
getviewsettings	returns information about the current viewport
setviewport	sets the current output viewport for graphics output

Image Manipulation

getimage	saves a bit image of the specified region to memory
imagesize	returns the number of bytes required to store a rectangular region of the screen
putimage	puts a previously-saved bit image onto the screen

Pixel Manipulation

getpixel	gets the pixel color at (x,y)
putpixel	plots a pixel at (x,y)

Besides drawing and painting, the graphics library offers several functions for manipulating the screen, viewports, images, and pixels. You can clear the whole screen in one fell swoop with a call to **cleardevice**; this routine erases the entire screen and homes the current position (CP) in the viewport, but leaves all other graphics system settings intact (the line, fill, and text styles, the palette, the viewport settings, etc.).

Depending on your graphics adapter, your system has between one and eight screen page buffers, which are areas in memory where individual whole-screen images are stored dot-by-dot. You can specify which screen page is the active one (where graphics functions place their output) and which is the visual page (the one displayed on screen) with **setactivepage** and **setvisualpage**, respectively.

Once your screen's in a graphics mode, you can define a viewport (a rectangular "virtual screen") on your screen with a call to **setviewport**. You define the viewport's position in terms of absolute screen coordinates, and specify whether clipping is *on* (active) or *off*. You clear the viewport with **clearviewport**. To find out the current viewport's absolute screen coordinates and clipping status, call **getviewsettings**.

You can capture a portion of the on-screen image with **getimage**, call **imagesize** to calculate the number of bytes required to store that captured image in memory, then put the stored image back on the screen (anywhere you want) with **putimage**.

The coordinates for all output functions (drawing, filling, text, etc.) are viewport-relative.

You can also manipulate the color of individual pixels with the functions **getpixel** (which returns the color of a given pixel) and **putpixel** (which plots a specified pixel in a given color).

Text Output in Graphics Mode

Here's a quick summary of the graphics-mode text output functions:

gettextsettings	returns the current text font, direction, size, and justification
outtext	sends a string to the screen at the current position (CP)
outtextxy	sends a string to the screen at the specified position
registerbgifont	registers a linked-in or user-loaded font for inclusion at link time
settextjustify	sets text justification values used by **outtext** and **outtextxy**
settextstyle	sets the current text font, style, and character magnification factor
setusercharsize	sets width and height ratios for stroked fonts
textheight	returns the height of a string in pixels
textwidth	returns the width of a string in pixels

The graphics library includes an 8×8 bit-mapped font and several stroked fonts for text output while in graphics mode.

- In a *bit-mapped* font, each character is defined by a matrix of pixels.
- In a *stroked* font, each character is defined by a series of vectors that tell the graphics system how to draw that character.

The advantage of using a stroked font is apparent when you start to draw large characters. Since a stroked font is defined by vectors, it will still retain good resolution and quality when the font is enlarged. On the other hand, when you enlarge a bit-mapped font, the matrix is multiplied by a scaling factor; as the scaling factor becomes larger, the characters' resolution becomes coarser. For small characters, the bit-mapped font should be sufficient, but for larger text you should select a stroked font.

You output graphics text by calling either **outtext** or **outtextxy**, and control the justification of the output text (with respect to the CP) with **settextjustify**. You select the character font, direction (horizontal or vertical), and size (scale) with **settextstyle**. You can find out the current text settings by calling **gettextsettings**, which returns the current text font, justification, magnification, and direction in a **textsettings** structure. **setusercharsize** allows you to modify the character width and height of stroked fonts.

If clipping is *on*, all text strings output by **outtext** and **outtextxy** will be clipped at the viewport borders. If clipping is *off*, these functions will throw away bit-mapped font output if any part of the text string would go off the screen edge; stroked font output is truncated at the screen edges.

The default 8×8 bit-mapped font is built in to the graphics package, so it is always available at run time. The stroked fonts are each kept in a separate .CHR file; they can be loaded at run time or converted to .OBJ files (with the BGIOBJ utility) and linked into your .EXE file.

To determine the on-screen size of a given text string, call **textheight** (which measures the string's height in pixels) and **textwidth** (which measures its width in pixels).

Normally, the **settextstyle** routine loads a font file by allocating memory for the font, then loading the appropriate .CHR file from disk. As an alternative to this dynamic loading scheme, you can link a character font file (or several of them) directly into your executable program file. You do this by first converting the .CHR file to an .OBJ file (using the BGIOBJ utility), then placing calls to **registerbgifont** in your source code (before the call to **settextstyle**) to *register* the character font(s). When you build your program, you need to link in the .OBJ files for the stroked fonts you register.

Note: Using **registerbgifont** is an advanced programming technique, not recommended for novice programmers. This function is described in more detail in Appendix D in this addendum.

Color Control

Here's a quick summary of the color control functions:

==

Get color information

getbkcolor	returns the current background color
getcolor	returns the current drawing color
getmaxcolor	returns the maximum color value available in the current graphics mode
getpalette	returns the current palette and its size

Set one or more colors

setallpalette	changes all palette colors as specified
setbkcolor	sets the current background color
setcolor	sets the current drawing color
setpalette	changes one palette color as specified by its arguments

==

Before summarizing how these color control functions work on CGA and EGA systems, we first present a basic description of how colors are actually produced on your graphics screen.

Pixels and Palettes

The graphics screen consists of an array of pixels; each pixel produces a single (colored) dot on the screen. The pixel's value does not specify the precise color directly; it is an index into a color table called a *palette.* The palette entry corresponding to a given pixel value contains the exact color information for that pixel.

This indirection scheme has a number of implications. Though the hardware might be capable of displaying many colors, only a subset of those colors can be displayed at any given time. The number of colors that can be displayed at any one time is equal to the number of entries in the palette (the palette's *size*). For example, on an EGA, the hardware can display 64 different colors, but only 16 of them at a time; the EGA palette's *size* = 16.

The *size* of the palette determines the range of values a pixel can assume, from 0 to (*size* – 1). The **getmaxcolor** function returns the highest valid pixel value (*size* – 1) for the current graphics driver and mode.

In this addendum, we often use the term *color*, such as the current drawing color, fill color and pixel color. In fact, this color is like a pixel value: it's an index into the palette. Only the palette determines the true color on the screen. By manipulating the palette, you can change the actual color displayed on the screen even though the pixel values (drawing color, fill color, etc.) have not changed.

Background and Drawing Color

The *background color* always corresponds to pixel value 0. When an area is cleared to the background color, that area's pixels are simply set to 0.

The *drawing color* is the value to which pixels are set when lines are drawn. You select a drawing color with `setcolor(n)`, where *n* is a valid pixel value for the current palette.

Color Control on a CGA

Due to graphics hardware differences, how you actually control color differs quite a bit between the CGA and the EGA, so we'll present them separately. Color control on the AT&T driver and the lower resolutions of the MCGA driver is similar to CGA color control.

On the CGA, you can choose to display your graphics in low resolution (320x200), which allows you to use four colors, or high resolution (640x200), in which you can use two colors.

CGA Low Resolution

In the low resolution modes, you can choose from four predefined four-color palettes. In any of these palettes, you can only set the first palette entry; entries 1, 2, and 3 are fixed. The first palette entry (color 0) is the background color. This background color can be any one of the 16 available colors (see following table).

You choose which palette you want by the mode you select (CGAC0, CGAC1, CGAC2, CGAC3): these modes use color palette 0 through color palette 3, as detailed in the following table.

Palette Number	*Color assigned to pixel value* 1	2	3
0	lightgreen	lightred	yellow
1	lightcyan	lightmagenta	white
2	green	red	brown
3	cyan	magenta	lightgray

The available CGA background colors, defined in GRAPHICS.H, are listed in the following table.

Numeric Value	*Symbolic Name*
0	BLACK
1	BLUE
2	GREEN
3	CYAN
4	RED
5	MAGENTA
6	BROWN
7	LIGHTGRAY
8	DARKGRAY
9	LIGHTBLUE
10	LIGHTGREEN
11	LIGHTCYAN
12	LIGHTRED
13	LIGHTMAGENTA
14	YELLOW
15	WHITE

To assign one of these colors to the CGA background color, use `setbkcolor(`*`color`*`)`, where *color* is one of the entries in the preceding table. Note that for CGA, this color is not a pixel value (palette index); it directly specifies the *actual* color to be put in the first palette entry.

CGA High Resolution

In high resolution mode (640x200), the CGA displays two colors: a black background and a colored foreground. Pixels can take on values of either 0 or 1. Because of a quirk in the CGA itself, the foreground color is actually what the hardware thinks of as its background color: you set it with the **setbkcolor** routine. (Strange but true.)

The colors available for the colored foreground are those listed in the preceding table. The CGA uses this color to display all pixels whose value equals 1.

The modes that behave in this way are CGAHI, MCGAMED, MCGAHI, ATT400MED, and ATT400HI.

CGA Palette Routines

Because the CGA palette is predetermined, you should not use the **setallpalette** routine on a CGA. Also, you should not use `setpalette(`*`index, actual_color`*`)`, except for *index* = 0. (This is an alternate way to set the CGA background color to *actual_color*.)

Color Control on the EGA and VGA

On the EGA, the palette contains 16 entries from a total of 64 possible colors, and each entry is user-settable.

You can retrieve the current palette with **getpalette**, which fills in a structure with the palette's size (16) and an array of the actual palette entries (the "hardware color numbers" stored in the palette). You can change the palette entries individually with **setpalette**, or all at once with **setallpalette**.

The default EGA palette corresponds to the 16 CGA colors, as given in the previous color table: black is in entry 0, blue in entry 1, ..., white in entry 15. There are constants defined in GRAPHICS.H that contain the corresponding hardware color values: these are EGA_BLACK, EGA_WHITE, etc. You can also get these values with **getpalette**.

The `setbkcolor(`*`color`*`)` routine behaves differently on an EGA than on a CGA. On an EGA, **setbkcolor** copies the actual color value that's stored in entry #*color* into entry #0.

As far as colors are concerned, the VGA driver behaves like the EGA driver; it just has higher resolution (and smaller pixels).

Error Handling in Graphics Mode

Here's a quick summary of the graphics-mode error-handling functions:

==

grapherrormsg	returns an error message string for the specified *errorcode*
graphresult	returns an error code for the last graphics operation that encountered a problem

==

If an error occurs when a graphics library function is called (such as a font requested with **settextstyle** not being found), an internal error code is set. You retrieve the error code for the last graphics operation that reported an error by calling **graphresult**. The following error return codes are defined:

error code	*graphics_errors* constant	corresponding error message string
0	grOk	No error
–1	grNoInitGraph	(BGI) graphics not installed (use **initgraph**)
–2	grNotDetected	Graphics hardware not detected
–3	grFileNotFound	Device driver file not found
–4	grInvalidDriver	Invalid device driver file
–5	grNoLoadMem	Not enough memory to load driver
–6	grNoScanMem	Out of memory in scan fill
–7	grNoFloodMem	Out of memory in flood fill
–8	grFontNotFound	Font file not found
–9	grNoFontMem	Not enough memory to load font
–10	grInvalidMode	Invalid graphics mode for selected driver
–11	grError	Graphics error
–12	grIOerror	Graphics I/O error
–13	grInvalidFont	Invalid font file
–14	grInvalidFontNum	Invalid font number
–15	grInvalidDeviceNum	Invalid device number

A call to `grapherrormsg(graphresult)` will return the error strings listed in the previous table.

The error return code accumulates, changing only when a graphics function reports an error. The error return code is reset to 0 only when **initgraph** executes successfully, or when you call **graphresult**. Therefore, if

you want to know which graphics function returned which error, you should store the value of **graphresult** into a temporary variable and then test it.

State Query

Here's a quick summary of the graphics mode state-query functions:

getarccoords	returns information about the coordinates of the last call to **arc** or **ellipse**
getaspectratio	returns the aspect ratio of the graphics screen
getbkcolor	returns the current background color
getcolor	returns the current drawing color
getfillpattern	returns the user-defined fill pattern
getfillsettings	returns information about the current fill pattern and color
getgraphmode	returns the current graphics mode
getlinesettings	returns the current line style, line pattern, and line thickness
getmaxcolor	returns the current highest valid pixel value
getmaxx	returns the current *x* resolution
getmaxy	returns the current *y* resolution
getmoderange	returns the mode range for a given driver
getpalette	returns the current palette and its size
getpixel	returns the color of the pixel at *x,y*
gettextsettings	returns the current text font, direction, size, and justification
getviewsettings	returns information about the current viewport
getx	returns the *x* coordinate of the current position (CP)
gety	returns the *y* coordinate of the current position (CP)

In each of Turbo C's graphics functions categories there is at least one state-query function. These functions are mentioned under their respective categories and also covered here. Each of the Turbo C graphics state-query functions is named **get**<*something*> (except in the error-handling category). Some of them take no argument and return a single value representing the requested information; others take a pointer to a structure defined in GRAPHICS.H, fill that structure with the appropriate information, and return no value.

The state-query functions for the graphics system control category are **getgraphmode** and **getmoderange**: the former returns an integer representing the current graphics driver and mode, and the latter returns the range of modes supported by a given graphics driver. **getmaxx** and

getmaxy return the maximum *x* and *y* screen coordinates for the current graphics mode.

The drawing and filling state-query functions are **getarccoords**, **getaspectratio**, **getfillpattern**, **getfillsettings**, and **getlinesettings**. **getarccoords** fills a structure with coordinates from the last call to **arc** or **ellipse**; **getaspectratio** tells the current mode's aspect ratio, which the graphics system uses to make circles come out round. **getfillpattern** returns the current user-defined fill pattern. **getfillsettings** fills a structure with the current fill pattern and fill color. **getlinesettings** fills a structure with the current line style (solid, dashed, etc.), line width (normal or thick), and line pattern.

In the screen- and viewport-manipulation category, the state-query functions are **getviewsettings**, **getx**, **gety**, and **getpixel**. When you have defined a viewport, you can find out its absolute screen coordinates and whether clipping is active by calling **getviewsettings**, which fills a structure with the information. **getx** and **gety** return the (viewport-relative) *x*- and *y*-coordinates of the CP (current position). **getpixel** returns the color of a specified pixel.

The graphics mode text-output function category contains one all-inclusive state-query function: **gettextsettings**. This function fills a structure with information about the current character font, the direction in which text will be displayed (horizontal or bottom-to-top vertical), the character magnification factor, and the text-string justification (both horizontal and vertical).

Turbo C's color-control function category includes three state-query functions. **getbkcolor** returns the current background color, and **getcolor** returns the current drawing color. **getpalette** fills a structure with the size of the current drawing palette and the palette's contents. **getmaxcolor** returns the highest valid pixel value for the current graphics driver and mode (palette *size* – 1).

C H A P T E R

2

Additions to TC.EXE

This chapter explains the additions to Turbo C's menus, hot keys, and editor. The original Options/Environment menu in Turbo C's Integrated Environment has been split into two menus (Environments and Directories) with some added options. The new hot key lets you switch back and forth between the Turbo C screen and the saved output screen. Finally, the editor changes affect tab settings, optimal fill, matching delimiter pairs, and editor command keys.

The (New) Options/Directories Menu

This new menu contains some items that were on the Options/Environment menu of Turbo C version 1.0 and two new menu items. The menu items from version 1.0 are Include directories, Library directories (which has been modified; it was singular in version 1.0), Output directory, and Turbo C directory. The two new menu items are Pick file name and Current pick file.

Refer to Chapter 2 in the *Turbo C User's Guide* for descriptions of Include directories, Output directory, and Turbo C directory. Descriptions of Library directories, Pick file name, and Current pick file follow.

Library directories

In Turbo C version 1.0, you could specify one library directory with the Library directory menu item. Now you can list multiple library directories, up to a maximum of 127 characters (including whitespace).

Use the following guidelines when entering library directories:

You must separate multiple directory pathnames with a semicolon (;).

Whitespace before and after the semicolon is allowed, but not required.

Relative and absolute pathnames are allowed, including pathnames relative to the logged position in drives other than the current one.

An Example:

```
c:\turboc\lib; c:\turboc\mylibs; a:newturbo\mathlibs; a:..\vidlibs
```

See Chapter 3 in this addendum for details on multiple library directories.

Pick file name

This item defines the name of a pick file to load. Entering a name here loads that pick file (if it exists) and defines where Turbo C will save the pick file when you exit. When you change the pick file name, Turbo C saves the current pick file before loading the new one.

If no pick file name is listed here, then Turbo C only writes a pick file if the Current pick file menu item contains a file name.

See Chapter 6 in this addendum for a discussion of pick files.

Current pick file

This menu item shows the file name and location of the current pick file, if there is one. This item is always disabled; it is for information only. Current pick file shows a file name when a default pick file is loaded or when you type one in with the Pick file name menu item. If you change the pick file name or exit the integrated environment, Turbo C stores the current pick list information in this listed pick file.

The (Modified) Options/Environment Menu

This menu, containing six items, is quite different from the version 1.0 Options/Environment menu. Three of the items on this menu (Backup source files, Edit auto save, and Zoomed windows) exist in version 1.0; only the second has been changed. The other three items on this menu are new in version 1.5: Config auto save, Tab size, and Screen size.

Backup source files

(*Same as version 1.0*) By default, Turbo C automatically creates a backup of the file in the editor when you do a File/Save; the backup file is FILENAME.BAK (where FILENAME is the name of the file in the editor). You can turn this backup feature *on* and *off* with this toggle.

Edit auto save

(*Was Auto save edit in version 1.0*) With this toggled to *on*, Turbo C automatically saves your file in the editor whenever you use Run or File/OS shell (if the file has been modified since the last time you saved it).

Config auto save

This is a new menu item. Normally, Turbo C saves the current configuration (writes it out to disk) only when you select Options/Store options. With Config auto save *on*, Turbo C also saves the file whenever you select Run or File/OS shell, or when you exit the integrated environment (if the configuration file has never been saved or has been at all modified since it was last saved).

With Config auto save *on*, if the configuration file has not yet been saved, Turbo C chooses a file name for the auto saved file. This is the name of the last configuration file you stored or retrieved, or TCCONFIG.TC (in the current directory) if you haven't loaded, retrieved, or saved a configuration file yet.

Zoomed windows

(*Same as version 1.0*) If your Turbo C integrated environment screen is set up with the Edit window and Message window both showing, selecting Zoomed windows...*on* zooms both windows to full screen, with the active window visible.

Use *F6* to switch from one window to the other, just as you do when both windows are showing.

To "unzoom" the windows (return to the setup where both windows are showing) just select Zoomed windows...*off*.

Tab size

This is a new menu item. When the editor Tab mode is *on* and you press the *Tab* key, the editor inserts a tab character in the file and the cursor jumps to the next tab stop. This menu item allows you to dictate how far apart the tab stops are; any number in the range 2 through 16 is allowed (the default is 8).

To change the tab size, select Tab size, type in the size you prefer, and press *Enter*. Voila! The editor redisplays all tabs in the size you selected. You can save this new tab size in your configuration file (select Store options from the Options menu).

Screen size

This is a new menu item. When you select Screen size, another menu appears; the items on this Screen size menu allow you to specify whether your integrated environment screen displays text in 25, 43, or 50 lines. One or two of these items is enabled, depending on the type of video adapter in your PC.

25 line standard display

This is the standard PC display: 25 lines by 80 columns. This menu item is always enabled; it's the only screen size available to systems with a Monochrome Display Adapter (MDA) or Color Graphics Adapter (CGA).

43 line EGA display

If your PC is equipped with an EGA, this menu item is enabled, as is 25 line standard display (but 50 line VGA display is disabled). Select 43 line EGA display to transform your text to 43 lines by 80 columns.

50 line VGA display

If your system includes a VGA, this menu item is enabled, along with 25 line standard display (but 43 line EGA display is disabled). Select 50 line VGA display to transform your text to 50 lines by 80 columns.

New Hot Key

Turbo C version 1.5 has a new hot key: *Alt-F5* ("flip to/from saved screen").

When you are using TC, you see one of two screens—the integrated environment screen itself or the *output screen.* The integrated environment screen is what you see when you edit, compile, link, and debug your

programs. The output screen is what you see when you run a Turbo C executable program or temporarily exit to DOS through the File/OS shell menu command. With some exceptions, Turbo C is able to continuously preserve the contents of this screen in a "saved output screen" buffer, updating it each time you select Run or File/OS shell. To view this saved screen, press *Alt-F5* (the "saved screen" hot key).

How Long Will Turbo C Save the Screen?

Under certain conditions, Turbo C preserves the saved screen's contents so that—when you select File/OS shell or run a program—the screen picks up where you left off. Whenever you run a program from the integrated environment or select File/OS shell, TC resets the video screen mode back to the mode that was in effect when you started TC from the DOS prompt (the "start-up mode"). There are two general cases that cause Turbo C to discard the contents of the buffer containing the saved output screen:

1. You do a compile or a link; the compiler and linker both use the area in memory where the saved screen is preserved.
2. The video mode of the screen when you started TC is incompatible with the mode of the saved output screen.

Changes to the Turbo C Editor

Turbo C's built-in interactive editor (in TC.EXE) contains a few new features.

- *Setting Tab Sizes:* You can now set tab sizes, from 2 to 16 columns per tab stop.
- *Optimal fill:* In Autoindent mode, the editor now optimally fills leading blank space with a combination of tab characters and spaces, to make smaller files.
- *Pair matching:* The editor will find matching pairs of various delimiters in your source code for you.
- *Editor key reassignment:* With TCINST, you can customize your own editor command keys.

This section of the addendum covers these new editor features. For a comprehensive explanation of the interactive editor, refer to Appendix A in the *Turbo C Reference Guide.*

Setting Tab Sizes

The new menu item Tab size on the Options/Environment menu allows you to dictate how far apart the editor tab stops are; any number in the range 2 through 16 is allowed (the default is 8).

To change the way tabs are displayed in a file, just change Tab size to the size you prefer, and the editor redisplays all tabs in that file in the size you selected. The new tab size setting is stored in your configuration file when you save it (select Store options from the Options menu).

Note: When the editor Tab mode is *off*, pressing the tab key inserts enough space characters to move the cursor to the next "soft" tab stop. Soft tab stops align with the first letter of each word in the line of text immediately above the current line.

Another Note: When you send a marked block of text from the editor to a file (or to PRN) with the *Ctrl-K W* command, the editor treats all tab characters as *hardware tabs* and writes (or prints) them "as is". This generally yields tab stops at every eighth column. However, when you send text from the editor to the printer with the *Ctrl-K P* command, the editor treats tab characters as *software tabs* and prints them as the appropriate number of space characters (equal to the tab size you selected with Tab size).

Autoindent and Optimal Fill

Autoindent is an editor feature you toggle *on* or *off* in one of two ways:

- When in the Edit window, type *Ctrl-O I* or *Ctrl-Q I*. (Simultaneously hold down the Control key *and* O or Q, then type I.)
- When in TCINST (the Turbo C Installation program), select Setup environment, then toggle Autoindent mode to *on*.

With Autoindent mode and Insert mode both *on*, the editor automatically indents a new line to align with the first character in the previous line.

Under certain conditions, the editor fills the leading blank space of the new, indented line with an optimal combination of tab characters and space characters. (An optimal combination is one that uses the least number of characters.) These are the conditions necessary for optimal filling to occur:

1. Autoindent mode, Insert mode, and Tab mode are all *on*.

2. You have just pressed *Enter* to move the cursor from the end of an indented line down to a new, blank line. (The editor inserts enough leading space characters to align the cursor below the first character of the line it just left.)
3. You have not moved the cursor off of that new line. (However, you can use the *Left* and *Right* arrow keys, the *Tab* key, the *Backspace* key, and the space bar to move the cursor horizontally on the new line.)
4. You type a character or command, or move to another line.

When this sequence occurs, the editor replaces the leading whitespace (or space and tab characters) in the new line with a combination of tab characters and space characters, yielding the same amount of leading space with fewer characters.

Examples

- Tab size in the Options/Environment menu is set to 8 (tab stops are in columns 1, 9, 17, 25, ...); Autoindent, Tab, and Insert modes are *on*; and the cursor is at the end of a line that begins at column 27.
  - You press *Enter* to insert a new line; the editor positions the cursor at column 27 in that new line.
  - Without moving the cursor, you type a character on the new line.
  - The editor fills the beginning of the new line with three tab characters (to column 25) and two space characters (to column 27) for a total of five inserted fill characters.
- If, in this same example, Tab size is set to 5 (tab stops in columns 1, 6, 11, 16, 21, 26, ...), the editor fills with five tab characters (to column 26) and one space character.
- Or if Tab size is set to 6 (tab stops 1, 7, 13, 19, 25, ...), and you move the cursor to column 18 before typing your first characters, the editor fills with two tab characters (to column 13) and five space characters (to column 18).

How to Turn Off Optimal Fill

With Autoindent mode and Insert mode *on* (but Tab mode *off*), the editor still indents the new line to align with the beginning of the previous line, but it does this by filling with space characters only (no tabs).

Pair Matching

There you are, debugging your source file that is full of functions, parenthesized expressions, nested comments, and a whole slew of other constructs that use delimiter pairs. In fact, your file is teeming with

- braces: `{` and `}`
- angle brackets: `<` and `>`
- parentheses: `(` and `)`
- square brackets: `[` and `]`
- comment markers: `/*` and `*/`
- double quotes: `"` and `"`
- single quotes: `'` and `'`

Finding the match to a particular brace can be tricky. Suppose you have a complicated expression with a number of nested subexpressions, and you want to make sure all the parentheses are properly balanced. Or say you're at the beginning of a function that stretches over several screens, and you want to jump to the end of that function. With Turbo C's handy pair-matching commands, the solution is at your fingertips. Here's what you do:

1. Place the cursor on the delimiter in question (for example, the opening brace of some function that stretches for a couple of screens).
2. To locate the mate to this selected delimiter, simply press *Ctrl-Q Ctrl-[*. (In the example given, the mate should be at the end of the function.)
3. The editor immediately moves the cursor to the delimiter that matches the one you had selected. If it moves to the one you had intended to be the mate, you know that the intervening code contains no unmatched delimiters of that type. If it highlights the wrong delimiter, you know there's trouble in River City; now all you need to do is track down the source of the problem.

A Few Details About Pair Matching

We've told you the basics of Turbo C's "Match Pair" commands; now you need some details about what you can and can't do with these commands, and notes about a few subtleties to keep in mind. This section covers the following points:

- There are actually two "Match Pair" editor commands: one for forward matching and the other for backward matching. The two commands are

Ctrl-Q Ctrl-[	Match pair (forward)
Ctrl-Q Ctrl-]	Match pair (backward)

- The way the editor searches for comment delimiters (/* and */) is slightly different from the other searches.
- If there is no mate for the delimiter you've selected, the editor doesn't move the cursor.

Two Commands for Directional Matching

Two "Match Pair" commands are necessary because some delimiters are not *directional*, while others are.

For example, suppose you tell the editor to find the match for an opening brace ({) or an opening square bracket ([ ). The editor knows that the matching delimiter can't be located *before* the one you've selected, so it searches forward for a match. Opening braces and opening square brackets are directional: the editor knows in which direction to search for the mate, so it doesn't matter which "Match Pair" command you give. Given either command, the editor still searches in the correct direction.

Similarly, if you tell the editor to find the mate to a closing brace (}) or a closing parenthesis ( )), it knows that the mate can't be located *after* the selected delimiter, so it automatically searches backward for a match. Again, because these delimiters are directional, it doesn't matter which "Match Pair" command you give: the editor always searches in the correct direction.

However, if you tell the editor to find the match for a double quote (") or a single quote ('), it doesn't automatically know which way to go. You must specify the search direction by giving the correct "Match Pair" command. If you give the command *Ctrl-Q Ctrl-[*, the editor searches forward for the match; if you give the command *Ctrl-Q Ctrl-]*, it searches backward for the match.

The following table summarizes the delimiter pairs, whether they imply search direction, and whether they are nestable. (Nestable delimiters are explained after this table.)

Delimiter Pair	*Direction Implied?*	*Are They Nestable?*
`{ }`	Yes	Yes
`( )`	Yes	Yes
`[ ]`	Yes	Yes
`< >`	Yes	Yes
`/* */`	Yes	Yes and No
`" "`	No	No
`' '`	No	No

Nestable Delimiters

What does *nestable* mean? Simply that, when searching for the mate to a directional delimiter, the editor keeps track of how many "delimiter levels" it enters and exits during the search.

This is best illustrated with some examples:

Search for match to square bracket or parenthesis:

```
   Matched pair                  Matched pair
         :                             :
     ....:....                .........:........
     :       :                :                :
     :       :                :                :
array[arr2[x]]                ( (x>0) && (y>0) )
          : :                   :   :    :   :
          : :                   :   :    :   :
          :.:                   :...:    :...:
           :                      :        :
           :                      :        :
     Matched pair           Matched pair   :
                                           :
                                     Matched pair
```

The Search for Comment Delimiters

Because comment delimiters are two-character delimiters, you must take care when highlighting one for a "Match Pair" search. In either case, the editor only recognizes the *first* of the two characters: the / part of a /* comment delimiter, or the * part of a */ delimiter. If you place the cursor on

the *second* character in either of these delimiters, the editor won't know what you're looking for, so it won't do any searching at all.

Also, as shown in the preceding table, comment delimiters are sometimes nestable, sometimes not ("Yes and No"). This is not a vagary or an inability to decide: It is a test dependent on multiple conditions. ANSI-compatible C programs cannot contain nested comments, but Turbo C provides an optional "Nested comments" feature (the menu item Nested comments in the Options/Compiler/Source menu) that you can toggle ON and OFF. This feature affects the nestability of comment delimiters when it comes to pair matching.

- If Nested comments is toggled *on*, the editor treats comment delimiters as nestable and keeps track of the delimiter levels it enters and exits in the search for a match.
- If Nested comments is toggled *off*, the editor does not treat comment delimiters as nestable; when a **/*** pair is selected, the first ***/** pair the editor finds is the match (and vice versa).

Note: If unmatched delimiters of the same type in comments, quotes, or conditional compilation sections fall between the matched pair, this affects the search.

Here are some examples to illustrate these differences:

Nested comments toggled ON--forward search with ^Q ^[:

```
/*  /*  /*  /*  here are some nested comments  */  */  */  */
:                                                            :
:.....Match Level Selected                                   :
                                                             :
                                        Match Level Found.....:
```

Note: Backward search from the "Found" */ will yield the "Selected" /* when Nested comments is toggled ON.

Nested comments toggled OFF--forward search with ^Q ^[:

```
/*  /*  /*  /*  here are some nested comments  */  */  */  */
:                                              :
:.....Match Level Selected                     :
                                               :
                                Match Level Found.....:
```

```
Nested comments toggled OFF--backward search with ^Q ^]:

   /*  /*  /*  /*  here are some nested comments  */  */  */  */
                        :                                     :
                        :.....Match Level Found               :
                                                              :
                                    Match Level Selected.....:
```

Editor Hot Key Assignment

Note: This feature is covered in detail in "The New TCINST" in this addendum, so we'll cover just the basics here.

Turbo C's interactive editor provides many editing functions, which are assigned to certain hot keys (or hot key combinations); these are explained in detail in Appendix A of the *Turbo C Reference Guide.*

TCINST is Turbo C's optional customization (or "installation") program: one of its menus allows you to assign the Turbo C editing functions to other hot keys, if you prefer. (This is known as "rebinding the keys".)

To change Turbo C's editor commands, follow this general procedure:

1. Load TCINST.EXE (at the DOS prompt, type `tcinst` and press *Enter*), then select the Editor commands menu. The **`Install Editor`** screen comes up, displaying three columns of text.
   - The first column (on the left) describes the editing functions available.
   - The second column lists the *Primary* keystrokes; what you press to invoke a particular editing function.
   - The third column lists the *Secondary* keystrokes; these are optional alternate keystrokes you can also press to invoke the same editing function.
2. The bottom lines of text in the **`Install Editor`** screen summarize the keys you use to change entries in the Primary and Secondary columns. Press *Enter* to enter the keystroke-editing mode, then use the *Left* and *Right* arrow keys to move the highlight bar to either the Primary or Secondary column.
3. Use the *Up* and *Down* arrow keys to highlight the editing command you intend to rekey.
4. Press *Enter* to select the highlighted editing command; the defined keystroke(s) for that command appears in a pop-up window.

5. Press *Backspace* to delete individual keystrokes from right to left in the pop-up window, or press *F3* to clear all defined keystrokes from the window.
6. Keystroke combinations come in three flavors: `WordStar-like`, `Ignore case`, and `Verbatim`. Press *F4* to cycle through these until the one you want is highlighted on the bottom line of the screen. Refer to "The New TCINST" in this addendum for more information about these three variations.
7. Type in the new defined keystrokes for that editing function (up to a maximum of six keystrokes). If you want to erase the last keystroke you assigned, press *Backspace*. If you want to abandon the new key assignments to that function, press *F2* to restore the originally-assigned keys, or *Esc* to restore them *and* leave the keystroke-editing mode.
8. Once you're satisfied with the new (or restored) key assignment(s) to a given function, press *Enter* to accept them.
9. When you've finished assigning keys (you've accepted the last modification), press *Esc* to leave the `Install Editor` screen and return to TCINST's main menu.

Note: If you override a standard Turbo C hot key, you will not be able to use that Turbo C shortcut while in the editor.

C H A P T E R

3

Changes to Command-Line Turbo C

To provide you with more power and choices in organizing your files and directories, Turbo C version 1.5 has extended and enhanced certain features. The compiler now

- supports multiple library directories
- provides extended syntax for the `-L`, `-I`, and `-D` command-line options

With the ability to specify multiple library directories, you can now put your custom and third-party library files in a separate directory that the compiler will search (instead of just in the current directory). With the extended command-line syntax, you have greater flexibility in naming directory paths and defining symbols.

In this chapter we cover the enhancements to command-line Turbo C (TCC.EXE): refer to Chapter 2 in this addendum for information about changes to Turbo C's integrated environment.

A Recap: In the original version (1.0) of TCC.EXE, you could do the following on the command line:

- specify multiple include directories by listing multiple `-I`*`dirname`* options (one per directory)
- specify the standard library directory with a single `-L`*`dirname`* option
- define multiple symbolic constants by listing multiple `-D`*`xxx`* options (one per define)

The New Turbo C: You can now direct Turbo C to search multiple directories for libraries. In hand with this, the syntax for the *library directories* (`-L`), *include directories* (`-I`), and *define symbols* (`-D`) command-line

options has been extended to allow multiple listings with a single option (this is known as"ganging" options).

Extended Syntax for These Options

The library directory option (Options/Directories/Library directory in TC and `-L` in TCC) has been enhanced to allow multiple directories. Additionally, TCC's syntax for the `-I` and `-D` command-line options has been extended to allow ganged entries (a feature previously available only in the Turbo C integrated environment).

In a nutshell, here's the revised syntax for these three TCC options:

Library directories:	`-L`*`dirname`*`[;`*`dirname`*`;...]`
Include directories:	`-I`*`dirname`*`[;`*`dirname`*`;...]`
Defines:	`-D`*`symbol`*`[=`*`string`*`][;`*`symbol`*`[=`*`string`*`];...]`

The parameter *dirname* used with `-L` and `-I` can be any directory path name.

The parameter *symbol* used with `-D` is an identifier. You can optionally give it a value (like this: `-Dtime=year` or `-Dfill=no` or `-Dmcopr=0`). If you don't assign a value to *symbol* (like this: `-Dxxx`), Turbo C will `#define` it to a single space character.

You can enter these multiple directories and defines on the command line in the following ways:

- You can "gang" multiple entries with a single –L, –I, or –D option, separating ganged entries with a semicolon, like this:

```
-Ldirname1;dirname2;dirname3 -Iinc1;inc2;inc3 -Dxxx;yyy=1;zzz=NO
```

- You can place more than one of each option on the command line, like this:

```
-Ldirname1 -Ldirname2 -Ldirname3 -Iinc1 -Iinc2 -Iinc3 -Dxxx -Dyyy=1 -Dzzz=NO
```

- You can mix ganged and multiple listings, like this:

```
-Ldirname1;dirname2 -Ldirname3 -Iinc1;inc2 -Iinc3 -Dxxx -Dyyy=1;zzz=NO
```

If you list multiple –L, –I, or –D options on the command line, the result is cumulative: the compiler will search all the directories listed, or define the specified constants, in order from left to right.

Note: The integrated environment (TC.EXE) now also supports multiple library directories (under the Options/Directories/Library directories

menu item), using the same "ganged entry" syntax as the Include directories and Defines menu items. Refer to the chapter "Additions to TC.EXE" in this addendum for more information.

Implicit vs. User-specified Library Files

Turbo C recognizes two types of library files: *implicit* and *user-specified* (also known as *explicit* library files).

- Implicit library files are the ones Turbo C automatically links in. These are the C*x*.LIB files, EMU.LIB or FP87.LIB, MATH*x*.LIB, and the start-up object files (C0*x*.OBJ).
- User-specified library files are the ones you explicitly list on the command line or in a project file; these are file names with a .LIB extension.

The Enhanced Library File-Search Algorithms

Turbo C version 1.0 searched for user-specified libraries only as they were specified (nowhere else), and it only searched for implicit libraries in a single library directory.

In version 1.5, the way Turbo C searches for library files has been extended; the new search algorithm is very similar to the way it searches for the `#include` files listed in your source code. To wit: If you put an `#include <somefile.h>` statement in your source code, Turbo C will search for SOMEFILE.H only in the specified include directories. If, on the other hand, you put an `#include "somefile.h"` statement in your code, Turbo C will search for SOMEFILE.H first in the current directory; if it does not find the header file there, it will then search in the specified include directories.

These are the new library file-search algorithms:

- *Implicit libraries:* Turbo C searches for implicit libraries only in the specified library directories; this is similar to `#include <somefile.h>`.
- *Explicit libraries:* Where Turbo C searches for explicit (user-specified) libraries depends in part on how you list the library file name.
    - If you list an explicit library file name with no drive or directory (like this: `mylib.lib`), Turbo C will search for that library in the current directory first. Then (if the first search was unsuccessful), it will look in the specified library directories; this is similar to `#include "somefile.h"`.

- If you list a user-specified library with drive and/or directory information (like this: `c:mystuff\mylib1.lib`), Turbo C will search *only* in the location you explicitly listed as part of the library path name, and not in the specified library directories.

The new version 1.5 library-search algorithm is upwardly compatible with the version 1.0 library search, which means that your code written under version 1.0 will work without problems in the new version.

Using –L and –I in Configuration Files

If you do not understand how to use TURBOC.CFG (the command-line configuration file) with TCC.EXE, refer to these sections in the *Turbo C User's Guide*: "The TURBOC.CFG File" in Chapter 3, and "Writing the Configuration File" in Chapter 1.

The `-L` and `-I` options you list on the command line take priority over those in the configuration file. How this works is described in "The TURBOC.CFG File" (see reference): the explanation of `-I` option priority given there now also applies to `-L` options.

An Example With Notes

Here is an example of using a TCC command line that incorporates multiple library directories (-L) and include directories (-I) options.

1. You are logged into C:\TURBOC, where TCC.EXE resides. Your A drive's current logged position is A:\ASTROLIB.
2. Your include files (.H or "header" files) are located in C:\TURBOC\INCLUDE.
3. Your startup files (C0T.OBJ, C0S.OBJ, ... , C0H.OBJ) are in C:\TURBOC\STARTUPS.
4. Your standard Turbo C library files (CS.LIB, CM.LIB, ..., MATHS.LIB, MATHM.LIB, ... , EMU.LIB, FP87.LIB, etc.) are in C:\TURBOC\LIB.
5. Your custom library files for star systems (which you created and manage with TLIB) are in C:\TURBOC\STARLIB. One of these libraries is PARX.LIB.
6. Your third-party-generated library files for quasars are in the A drive, in A:\ASTROLIB; one of these libraries is WARP.LIB.

Under this configuration you enter the following TCC command line:

```
tcc -mm -Lstartups;lib;starlib -Iinclude orion umaj parx.lib a:\astrolib\warp.lib
```

TCC will compile ORION.C and UMAJ.C to .OBJ files, then link them with the medium model start-up code (C0M.OBJ), the medium model libraries (CM.LIB, MATHM.LIB), the standard floating-point emulation library (EMU.LIB), and the user-specified libraries (PARX.LIB and WARP.LIB), producing an executable file named ORION.EXE.

The compiler will search C:\TURBOC\INCLUDE for the include files in your source code.

It will search for the startup code in C:\TURBOC\STARTUPS (then stop because they're there); it will search for the standard libraries in C:\TURBOC\STARTUPS (not there) then in C:\TURBOC\LIB (search ends because they're there).

When searching for the user-specified library PARX.LIB, the compiler first looks in the current directory, C:\TURBOC. Not finding the library there, the compiler then searches the library directories in order: first C:\TURBOC\STARTUPS, then C:\TURBOC\LIB, then C:\TURBOC\STARLIB (where it locates PARX.LIB).

For the library WARP.LIB, an explicit path is given (A:\ASTROLIB\WARP.LIB), so the compiler only looks there.

C H A P T E R

4

New and Modified Functions and Variables

The information in this chapter is meant to supplement the global variable and function lookup sections (Chapters 1 and 2) of your Turbo C Reference Guide.

New and Modified Global Variables

These descriptions of global variables supplement Chapter 1 in your Turbo C Reference Guide.

_argc, _argv — *new*

Names *_argc* – count of command-line arguments
_argv – array of command-line arguments

Usage extern int *_argc;*
extern char ***_argv;*

Declared in dos.h

Description *_argc* has the value of *argc* passed to **main()** when the program started.

_argv points to an array containing the original command-line arguments (the elements of *argv[ ]*) passed to **main()** when the program started.

directvideo — *new*

Name *directvideo* – direct output to video RAM flag

Usage extern int *directvideo;*

Declared in conio.h

Description *directvideo* controls whether your program's console output goes directly to the video RAM (*directvideo* = 1) or goes via ROM BIOS calls (*directvideo* = 0).

The default value is *directvideo* = 1 (console output goes directly to video RAM). In order to use *directvideo* = 1, your system's video hardware must be identical to IBM display adapters. Setting *directvideo* = 0 allows your

console output to work on any system that is IBM BIOS-compatible.

_heaplen, _stklen — *modified*

Names

_heaplen – heap length variable
_stklen – stack length variable

Usage

extern unsigned *_heaplen;*
extern unsigned *_stklen;*

Declared in

dos.h

Description

_heaplen specifies the size of the near heap in the small data models (tiny, small, and medium). *_heaplen* does not exist in the large data models (compact, large, and huge) as they do not have a near heap.

_stklen specifies the size of the stack for all six memory models. The minimum stack size allowed is 128 words; if you give a smaller value, *_stklen* is automatically adjusted to the minimum. The default stack size is 4K.

In the small and medium models, the data segment size is computed as follows:

```
data segment [small, medium] = global data + heap + stack
```

If *_heaplen* is set to 0, the program allocates 64K bytes for the data segment and the effective heap size is

```
64K - (global data + stack) bytes.
```

By default, *_heaplen* = 0, so you'll get a 64K data segment unless you specify a particular *_heaplen* value.

In the tiny model, everything (including code) is in the same segment, so the data segment computations are adjusted to include the code plus 256 bytes for the Program Segment Prefix.

```
data segment[tiny] = 256 + code + global data + heap + stack
```

If *_heaplen* = 0 in the tiny model, the effective heap size is obtained by subtracting the PSP, code, global data and stack from 64K.

In the compact and large models, there is no near heap, so the data segment is simply:

```
data segment [compact, large] = global data + stack
```

In the huge model, the stack is a separate segment, and each module has its own data segment.

_8087 — *modified*

Name _8087 – coprocessor chip flag

Usage extern int _8087;

Declared in dos.h

Description The _8087 variable is set to 1 if the start-up code auto-detection logic detects a floating-point coprocessor (an 8087, 80287, or 80387), or if the 87 environment variable is set to **Y** (`SET 87=Y`). The _8087 variable is set to 0 otherwise.

(Refer to Chapter 9 in the *Turbo C User's Guide* for more information about the 87 environment variable.)

You must have floating-point code in your program for the _8087 variable to be set to 1.

New and Modified Functions

These descriptions of functions supplement Chapter 2 in your Turbo C Reference Guide. *Most of the functions described here are new, though a few of these entries give updated information about functions described in the reference guide.*

arc *graphics*

Name arc – draws a circular arc

Usage

```
#include <graphics.h>
void far arc(int x, int y, int stangle, int endangle,
             int radius);
```

Related functions usage

```
void far circle(int x, int y, int radius);
void far ellipse(int x, int y, int stangle, int endangle,
                 int xradius, int yradius);

void far getarccoords(struct arccoordstype
                      far *arccoords);

void far getaspectratio(int far *xasp, int far *yasp);
void far pieslice(int x, int y, int stangle, int endangle,
                  int radius);
```

Prototype in graphics.h

Description Each of the four draw functions described here (**arc**, **circle**, **ellipse**, and **pieslice**) draws the outline of its shape in the current drawing color.

arc draws a circular arc centered at (*x*,*y*) with a radius given by *radius*. The **arc** travels from *stangle* to *endangle*. If *stangle* = 0 and *endangle* = 360, the call to **arc** will draw a complete circle.

circle draws a circle, with its center at (*x*,*y*) and a radius given by *radius*.

ellipse draws an elliptical arc, with its center at (*x,y*) and the horizontal and vertical axes given by *xradius* and *yradius*, respectively. The ellipse travels from *stangle* to *endangle*. If *stangle* = 0 and *endangle* = 360, the call to **ellipse** will draw a complete ellipse.

pieslice draws and fills a pie slice centered at (*x,y*) with a radius given by *radius*. The slice travels from *stangle* to *endangle*. The slice is outlined in the current drawing color and then filled using the current fill pattern and fill color.

The angles for **arc**, **ellipse**, and **pieslice** are counterclockwise, with 0 degrees at 3 o'clock, 90 degrees at 12 o'clock, etc.

Each graphics driver and graphics mode has an aspect ratio associated with it. The aspect ratio is used by the **arc**, **circle**, and **pieslice** routines as a scaling factor to make circles round on the screen. This ratio can computed by calling **getaspectratio**, then manipulating **xasp* and **yasp*.

The *y* aspect factor, **yasp*, is normalized to 10,000; on all graphics adapters except the VGA, **xasp* (the *x* aspect factor) is less than **yasp* because the pixels are taller than they're wide. On the VGA, which has "square" pixels, **xasp* = **yasp*. In general, the relationship between **yasp* and **xasp* can be stated as:

**yasp* = 10,000
**xasp* < = 10,000

getarccoords fills in the **arccoordstype** structure pointed to by *arccoords* with information about the last call to **arc**. The **arccoordstype** structure is defined in GRAPHICS.H as follows:

```
struct arccoordstype {
   int x, y;
   int xstart, ystart, xend, yend;
};
```

The members of this structure are used to specify the center point (*x,y*), the starting position (*xstart, ystart*), and the ending position (*xend, yend*) of the arc. These values are useful if you need to make a line meet at the end of an arc.

Return value If an error occurs while filling the pie slice, **graphresult** will return a value of –6.

Portability Similar routines exist in Turbo Pascal 4.0.

See also **getfillsettings**

Example

```
#include <graphics.h>

main()
{
   int graphdriver = DETECT, graphmode;        /* will request autodetection */

   struct arccoordstype arcinfo;
   int xasp, yasp;
   long xlong;

   initgraph(&graphdriver, &graphmode, "");            /*initialize graphics */

/* Draw a 90 degree arc with radius of 50   */
   arc(150, 150, 0, 89, 50);

/* Get the coordinates of the arc and connect ends */
   getarccoords(&arcinfo);
   line(arcinfo.xstart, arcinfo.ystart, arcinfo.xend, arcinfo.yend);

/* Draw a circle */
   circle(150, 150, 100);

/* Draw an ellipse inside the circle */
   ellipse(150, 150, 0, 359, 100, 50);

/* Draw and fill a pieslice */
   setcolor(WHITE);                                        /* white outline */
   setfillstyle(SOLID_FILL, LIGHTRED);
   pieslice(100, 100,   0, 134, 49);
   setfillstyle(SOLID_FILL, LIGHTBLUE);
   pieslice(100, 100, 135, 225, 49);
   setfillstyle(SOLID_FILL, WHITE);
   pieslice(100, 100, 225, 360, 49);

/* Draw a "square" rectangle */
   getaspectratio(&xasp, &yasp);
   xlong = (100L * (long)yasp) / (long)xasp;
   rectangle(0, 0, (int)xlong, 100);

   closegraph();
}
```

Name **assert** – tests a condition and possibly aborts

Usage
#include <assert.h>
#include <stdio.h>
void assert(int *test*);

Prototype in assert.h

Description **assert** is a macro that expands to an `if` statement; if *test* in the expanded macro fails, **assert** prints a message and aborts the program (via a call to **abort**).

The message **assert** prints is:

`Assertion failed: <test>, file <filename>, line <linenum>`

The *filename* and *linenum* listed in the message are the source file name and line number where the **assert** macro appears.

If you place the `#define NDEBUG` directive ("no debugging") in the source code before the `#include <assert.h>` directive, the effect is to comment out the **assert** statement.

Return value None

Portability This macro is available on some UNIX systems, including Systems III and V.

See also **abort**

Example

```
/* ASSERTST.C: add an item to a list, verify that the item is not NULL */

#include <assert.h>
#include <stdio.h>

struct ITEM {
   int key;
   int value;
};

void additem(struct ITEM *itemptr)
{
   assert(itemptr != NULL);                          /* this is line 13 */
   /* ... add the item ... */
}

main()
```

```
{
   additem(NULL);
}
```

Program Output

```
Assertion failed: itemptr != NULL, file C:\TURBOC\ASSERTST.C, line 13
```

bar *graphics*

Name **bar** – draws a bar

Usage #include <graphics.h>
void far bar(int *left*, int *top*, int *right*, int *bottom*);

Related functions usage void far bar3d(int *left*, int *top*, int *right*, int *bottom*, int *depth*, int *topflag*);

Prototype in graphics.h

Description **bar** draws a filled-in rectangular bar. The bar is filled using the current fill pattern and fill color. **bar** does not outline the bar; to outline a two-dimensional bar, use **bar3d** with *depth* = 0.

bar3d draws a three-dimensional rectangular bar, then fills it in using the current fill pattern and fill color. The 3-D outline of the bar is drawn in the current line style and color. The bar's depth, in pixels, is given by *depth*. The *topflag* parameter governs whether or not a 3-D top is put on the bar. If *topflag* is non-zero, a top is put on; otherwise, no top is put on the bar (making it possible to stack several bars on top of one another).

In both functions, the upper-left and lower-right corners of the rectangle are given by (*left*,*top*) and (*right*,*bottom*), respectively.

To calculate a typical depth for **bar3d**, take 25% of the width of the bar, like this:

```
bar3d(left, top, right, bottom, (right - left)/4, 1);
```

Return value None

Portability Similar routines exist in Turbo Pascal 4.0

See also **getbkcolor, getfillsettings, getlinestyle, graphresult, rectangle**

Example

```
#include <graphics.h>

main()
{
   int graphdriver = DETECT, graphmode;        /* will request autodetection */

   initgraph(&graphdriver, &graphmode, "");          /* initialize graphics */

   setfillstyle(SOLID_FILL, MAGENTA);
   bar3d(100, 10, 200, 100, 5, 1);
   setfillstyle(HATCH_FILL, RED);
   bar(30, 30, 80, 80);

   closegraph();
};
```

bar3d *graphics*

Name **bar3d** – draws a 3-D bar

Usage #include <graphics.h>
void far bar3d(int *left*, int *top*, int *right*, int *bottom*,
int *depth*, int *topflag*);

Prototype in graphics.h

Description see **bar**

Name **bsearch** – binary search

Usage

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t nelem, size_t width,
              int (*fcmp)(const void *, const void *));
```

Related functions usage

```
void *lfind(const void *key, const void *base,
            size_t *pnelem, size_t width,
            int (*fcmp)(const void *, const void *));

void *lsearch(const void *key, void *base,
              size_t *pnelem, size_t width,
              int (*fcmp)(const void *, const void *));
```

Prototype in stdlib.h

Description These functions have the same description as given in the *Turbo C Reference Guide*, with the following exceptions:

Revised arguments in prototypes:

The type *size_t* is defined with `typedef` to be an unsigned integer.

- *nelem* gives the number of elements in the table (**bsearch** only)
- *pnelem* points to the number of elements in the table (**lfind** and **lsearch** only)
- *width* specifies the number of bytes in each table entry

New description of the comparison routine:

fcmp*, the comparison routine, is called with two arguments, *elem1* and *elem2*. Each argument points to an item to be compared. The comparison function compares each of the pointed-to items (elem1* and **elem2*), and returns an integer based on the results of the comparison. Typically, *elem1* is the argument *key*, and *elem2* is a pointer to an element in the table being searched.

Return value These functions return the same values as given in the *Turbo C Reference Guide.*

New description for return from comparison routine:

For **bsearch**, the **fcmp* return value is

< 0	if	**elem1*	<	**elem2*
== 0	if	**elem1*	==	**elem2*
> 0	if	**elem1*	>	**elem2*

For **lsearch** and **lfind**, only equality matters, so the **fcmp* return value is

== 0	if	**elem1* == **elem2*
!= 0	if	**elem1* is different from **elem2*

calloc *modified*

Name **calloc** – allocates main memory

Usage

```
#include <stdlib.h>
void *calloc(size_t nelem, size_t elsize);
```

Declared in stdlib.h, alloc.h

Description see **malloc** (in this addendum and in the *Turbo C Reference Guide*)

chsize *misc*

Name **chsize** – changes file size

Usage int chsize(int *handle*, long *size*);

Prototype in io.h

Description **chsize** changes the size of the file associated with *handle*. It can truncate or extend the file, depending on the value of *size* compared to the file's original size.

The mode in which you open the file must allow writing.

If **chsize** extends the file, it will append null characters (\0). If it truncates the file, all data beyond the new end-of-file indicator is lost.

Return value	On success, **chsize** returns 0. On failure, it returns –1 and *errno* is set to one of the following:

EACCESS	Permission denied
EBADF	Bad file number

Portability	Unique to MS-DOS.
See also	**creat, fopen**

circle *graphics*

Name	**circle** – draws a circle
Usage	#include <graphics.h> void far circle(int *x*, int *y*, int *radius*);
Prototype in	graphics.h
Description	see **arc**

cleardevice *graphics*

Name	**cleardevice** – clears the graphics screen
Usage	#include <graphics.h> void far cleardevice(void);
Prototype in	graphics.h
Description	**cleardevice** erases the entire graphics screen and moves the CP (current position) to home (0,0).
Return value	None
Portability	A similar routine exists in Turbo Pascal 4.0
See also	**clearviewport**

clearviewport *graphics*

Name	**clearviewport** – clears the current viewport
Usage	#include <graphics.h> void far clearviewport(void);
Prototype in	graphics.h

Description — **clearviewport** erases the viewport and moves the CP (current position) to home (0,0).

Return value — None

Portability — A similar routine exists in Turbo Pascal 4.0

See also — **getviewsettings, cleardevice**

Example

```
setviewport(30, 30, 130, 130, 0);
outtextxy(10, 10, "Hit any key to clear viewport ...");
getch();                                              /* get a key */
clearviewport();                        /* clear viewport when key is hit */
```

closegraph — *graphics*

Name — **closegraph** – shuts down the graphics system

Usage — #include <graphics.h>
void far closegraph(void);

Prototype in — graphics.h

Description — see **initgraph**

clreol — *text*

Name — **clreol** – clears to end of line in text window

Usage — void clreol(void);

Prototype in — conio.h

Description — **clreol** clears all characters from the cursor position to the end of the line within the current text window without moving the cursor.

Return value — None

Portability — This function works with IBM PCs and compatibles, only; a corresponding function exists in Turbo Pascal.

See also — **clrscr, delline, window**

clrscr *text*

Name **clrscr** – clears text mode window

Usage void clrscr(void);

Prototype in conio.h

Description **clrscr** clears the current text window and places the cursor in the upper left-hand corner (at position 1,1).

Return value None

Portability This function works with IBM PCs and compatibles, only; a corresponding function exists in Turbo Pascal.

See also **clreol, delline, window**

country *modified*

Name **country** – returns country-dependent information

Usage #include <dos.h>
struct country *country (int *countrycode*,
struct country **countryp*);

Prototype in dos.h

Description The description of **country** in the *Turbo C Reference Guide* is correct except for the definition of the structure **country**; this is the updated definition of that structure:

```
struct country {
   int co_date;                              /* date format */
   char co_curr[5];                      /* currency symbol */
   char co_thsep[2];                 /* thousands separator */
   char co_desep[2];                   /* decimal separator */
   char co_dtsep[2];                      /* date separator */
   char co_tmsep[2];                     /* time separator  */
   char co_currstyle;                     /* currency style */
   char co_digits;    /* # of signif. digits in currency */
   char co_time;                             /* time format */
   long co_case;                                /* case map */
   char co_dasep[2];                     /* data separator  */
   char co_fill[10];                              /* filler */
};
```

cprintf *modified*

Name	**cprintf** – sends formatted output to the screen
Usage	int cprintf(const char **format*[, *argument*, ...]);
Prototype in	conio.h
Description	**cprintf** has been modified so output is written to the current text window. (See the *Turbo C Reference Guide* for further description.)
Return value	**cprintf** returns the number of bytes output.
Portability	This function works with IBM PCs and compatibles only.

cputs *modified*

Name	**cputs** – sends a string to the screen
Usage	int cputs(const char * *string*);
Prototype in	conio.h
Description	**cputs** has been modified so output is written to the current text window. (See the *Turbo C Reference Guide* for further description.)
Return value	**cputs** returns the last character printed.
Portability	This function works with IBM PCs and compatibles only.

delay *misc*

Name	**delay** – suspends execution for interval (milliseconds)
Usage	void delay(unsigned *milliseconds*);
Prototype in	dos.h
Description	With a call to **delay**, the current program is suspended from execution for the number of milliseconds specified by the argument *milliseconds*. The exact time may vary somewhat in different operating environments.
Return value	None

Portability This function works with IBM PCs and compatibles only; a corresponding function exists in Turbo Pascal.

See also **sleep, sound**

Example

```
/* emits a 440 Hz tone for 500 milliseconds */

main()
{
   sound(440);
   delay(500);
   nosound();
}
```

delline *text*

Name **delline** – deletes line in text window

Usage void delline(void);

Prototype in conio.h

Description **delline** deletes the line containing the cursor and moves all lines below it one line up. **delline** operates within the currently active text window.

Return value None

Portability This function works with IBM PCs and compatibles, only; a corresponding function exists in Turbo Pascal.

See also **clreol, insline, window**

detectgraph *graphics*

Name **detectgraph** – determines graphics driver and mode to use by checking the hardware

Usage #include <graphics.h>
void far detectgraph(int far **graphdriver*,
int far **graphmode*);

Prototype in graphics.h

Description see **initgraph**

Name — **div** – divide two integers, returning quotient and remainder

Usage

#include <stdlib.h>
div_t div(int *numer*, int *denom*);

Related functions usage

ldiv_t ldiv(long *lnumer*, long *ldenom*);

Prototype in — stdlib.h

Description

div divides two integers and returns both the quotient and the remainder as a *div_t* type. *numer* and *denom* are the numerator and denominator, respectively. The *div_t* type is a structure of integers defined (with **typedef**) in STDLIB.H as follows:

```
typedef struct {
   int  quot;                          /* quotient */
   int  rem;                          /* remainder */
} div_t;
```

ldiv divides two longs and returns both the quotient and the remainder as an *ldiv_t* type. *lnumer* and *ldenom* are the numerator and denominator, respectively. The *ldiv_t* type is a structure of longs defined (with **typedef**) in STDLIB.H as follows:

```
typedef struct {
   long quot;                          /* quotient */
   long rem;                          /* remainder */
} ldiv_t;
```

Return value — Each function returns a structure whose elements are *quot* (the quotient) and *rem* (the remainder).

Portability — ANSI C

Example

```
#include <stdlib.h>
div_t x;
ldiv_t lx;

main()
{
   x = div(10,3);
```

```
    printf("10 div 3 = %d remainder %d\n", x.quot, x.rem);

    lx = ldiv(100000L, 30000L);
    printf("100000 div 30000 = %ld remainder %ld\n", lx.quot, lx.rem);
}
```

drawpoly *graphics*

Name **drawpoly** – draws the outline of a polygon

Usage #include <graphics.h>
void far drawpoly(int *numpoints*, int far **polypoints*);

Related functions usage void far fillpoly(int *numpoints*, int far **polypoints*);

Prototype in graphics.h

Description **drawpoly** draws a polygon with *numpoints* points, using the current line style and color.

fillpoly draws the outline of a polygon in the current line style and color (just as **drawpoly** does), then fills the polygon using the current fill style and fill color.

polypoints points to a sequence of (*numpoints* * 2) integers. Each pair of integers gives the x and y coordinates of a point on the polygon.

Note: In order to draw a closed figure with n vertices, you must pass $n + 1$ coordinates to **drawpoly** where the nth coordinate is equal to the 0th.

Return value If an error occurs while filling the polygon, **graphresult** will return a value of –6.

Portability Similar routines exist in Turbo Pascal 4.0

See also **getfillsettings, getlinesettings, getbkcolor, graphresult**

Example

```
#include <graphics.h>

main()
{
   int graphdriver = DETECT, graphmode;        /* will request autodetection */

   int triangle[ ] = {50,100, 100,100, 150,150, 50,100};
   int rhombus[ ] = {50,10, 90,50, 50,90, 10,50};

   initgraph(&graphdriver, &graphmode, "");          /* initialize graphics */
```

```
    /* draw a triangle */
    drawpoly(sizeof(triangle)/(2*sizeof(int)), triangle);

    /* draw and fill a rhombus */
    fillpoly(sizeof(rhombus)/(2*sizeof(int)), rhombus);

    closegraph();
};
```

ellipse *graphics*

Name **ellipse** – draws an elliptical arc

Usage #include <graphics.h>
void far ellipse(int *x*, int *y*, int *stangle*, int *endangle*,
int *xradius*, int *yradius*);

Prototype in graphics.h

Description see **arc**

exec... *modified*

Name **exec...** – functions that load and run other programs

Usage Refer to *Turbo C Reference Guide*

Prototypes in process.h

Description These functions have the same description as given in the *Turbo C Reference Guide*, with the following exception:

The description (given in the *Turbo C Reference Guide*) of how **exec...** functions search for files is not complete; the **exec...** functions search for *pathname* as follows.

- If no explicit extension is given (for example, *pathname* = `MYPROG`), the functions will search for the file as given. If that one is not found, they will add .COM and search again. If that's not found, they'll add .EXE and search one last time.
- If an explicit extension or period is given (for example, *pathname* = `MYPROG.EXE`), the functions will search for the file as given.
- For the **exec...** functions with a *p* suffix, if *pathname* does not contain an explicit directory, the functions

will search first the current directory, then the directories set with the DOS PATH environment variable.

fgetpos *misc*

Name **fgetpos** – gets the current file pointer

Usage #include <stdio.h>
int fgetpos(FILE **stream*, fpos_t **pos*);

Related functions usage int fsetpos(FILE **stream*, const fpos_t **pos*);

Prototype in stdio.h

Description **fgetpos** stores the position of the file pointer associated with *stream* in the location pointed to by *pos*.

fsetpos sets the file pointer associated with *stream* to a new position. The new position is the value obtained by a previous call to **fgetpos** on that stream. **fsetpos** clears the end-of-file indicator on the file that *stream* points to, plus undoes any effects of **ungetc** on that file. After a call to **fsetpos**, the next operation on the file can be input or output.

The type *fpos_t* is defined in STDIO.H as

```
typedef long fpos_t;
```

Return value On success, **fgetpos** and **fsetpos** return 0. On failure, both functions return a non-zero value.

See also **fseek**

fillpoly *graphics*

Name **fillpoly** – draws and fills a polygon

Usage #include <graphics.h>
void far fillpoly(int *numpoints*, int far **polypoints*);

Prototype in graphics.h

Description see **drawpoly**

Name	**floodfill** –flood-fills a bounded region
Usage	#include <graphics.h> void far floodfill(int *x*, int *y*, int *border*);
Prototype in	graphics.h
Description	**floodfill** fills an enclosed area on bitmap devices. (*x,y*) is a "seed point" within the enclosed area to be filled. The area bounded by the color *border* is flooded with the current fill pattern and fill color. If the seed point is within an enclosed area, then the inside will be filled. If the seed is outside the enclosed area, then the exterior will be filled. Use **fillpoly** instead of **floodfill** whenever possible so that you can maintain code compatibility with future versions.
Return value	If an error occurs while flooding a region, **graphresult** will return a value of –7.
Portability	A similar routine exists in Turbo Pascal 4.0
See also	**drawpoly,getbkcolor, getfillsettings, getlinesettings, graphresult**

Example

```
#include <graphics.h>

main()
{
   int graphdriver = DETECT, graphmode;        /* will request autodetection */

   initgraph(&graphdriver, &graphmode, "");         /* initialize graphics */

/* Draw a bar, then flood-fill the side and top  */
   setcolor(WHITE);
   setfillstyle(HATCH_FILL, LIGHTMAGENTA);
   bar3d(10, 10, 100, 100, 10, 1);
   setfillstyle(SOLID_FILL, LIGHTGREEN);
   floodfill(102, 50, WHITE);                             /* fill the side */
   floodfill(50, 8, WHITE);                                /* fill the top */

   closegraph();
};
```

fsetpos *misc*

Name — **fsetpos** – positions the file pointer on a stream

Usage — #include <stdio.h>
int fsetpos(FILE **stream*, const fpos_t **pos*);

Prototype in — stdio.h

Description — see **fgetpos**

getarccoords *graphics*

Name — **getarccoords** – gets coordinates of the last call to **arc**

Usage — #include <graphics.h>
void far getarccoords(struct arccoordstype
far **arccoords*);

Prototype in — graphics.h

Description — see **arc**

getaspectratio *graphics*

Name — **getaspectratio** – returns the current graphics mode's aspect ratio

Usage — #include <graphics.h>
void far getaspectratio(int far **xasp*, int far **yasp*);

Prototype in — graphics.h

Description — see **arc**

getbkcolor *graphics*

Name — **getbkcolor** – returns the current background color

Usage — #include <graphics.h>
int far getbkcolor(void);

Related functions usage — void far setbkcolor(int *color*);

Prototype in — graphics.h

Description

getbkcolor returns the current background color. (See following table for details.)

setbkcolor sets the background to the color specified by *color*. The argument *color* can be a name or a number, as listed in the following table:

Number	*Name*	*Number*	*Name*
0	BLACK	8	DARKGRAY
1	BLUE	9	LIGHTBLUE
2	GREEN	10	LIGHTGREEN
3	CYAN	11	LIGHTCYAN
4	RED	12	LIGHTRED
5	MAGENTA	13	LIGHTMAGENTA
6	BROWN	14	YELLOW
7	LIGHTGRAY	15	WHITE

Note: These symbolic names are defined in GRAPHICS.H.

For example, if you want to set the background color to blue, you can call

```
setbkcolor(BLUE)
  /*   or   */
setbkcolor(1)
```

On CGA and EGA systems, **setbkcolor** changes the background color by changing the first entry in the palette.

Note: If you use an EGA or a VGA and you change the palette colors with **setpalette** or **setallpalette**, the defined symbolic constants might not give you the correct color.

Return value

getbkcolor returns the current background color.

setbkcolor returns nothing.

Portability

Similar routines exist in Turbo Pascal 4.0

See also

getpalette, initgraph

Example

```
#include <graphics.h>

main()
{
```

```
    int graphdriver = DETECT, graphmode;         /* will request autodetection */
    int svcolor;

    initgraph(&graphdriver, &graphmode, "");           /* initialize graphics */

    svcolor = getbkcolor();                         /* save current bk color */
    setbkcolor(svcolor ^ 1);                              /* change bk color */
    delay(5000);                                           /* wait 5 seconds */
    setbkcolor(svcolor);                           /* restore old bk color */

    closegraph();
};
```

getche *modified*

Name **getche** – gets character from keyboard, echoes to screen

Usage int getche(void);

Prototype in conio.h

Description **getche** has been modified so input is echoed to the current text window. (See the *Turbo C Reference Guide* for further description.)

getcolor *graphics*

Name **getcolor** – returns the current drawing color

Usage #include <graphics.h>
int far getcolor(void);

Related functions usage void far setcolor(int *color*);

Prototype in graphics.h

Description **getcolor** returns the current drawing color.

setcolor sets the current drawing color to *color*, which can range from 0 to `getmaxcolor()`.

The drawing color is the value to which pixels are set when lines, etc., are drawn. For example, in CGAC0 mode, the palette contains four colors: the background color, light green, light red, and yellow. In this mode, if `getcolor()` returns 1, the current drawing color is light green; similarly, `setcolor(3)` selects a drawing color of yellow.

Return value **getcolor** returns the current drawing color. **setcolor** returns nothing.

Portability Similar routines exist in Turbo Pascal 4.0

See also **getpalette, getmaxcolor**

Example

```
#include <graphics.h>

main()
{
   int graphdriver = DETECT, graphmode;        /* will request autodetection */
   int svcolor;

   initgraph(&graphdriver, &graphmode, "");          /* initialize graphics */

   svcolor = getcolor();                     /* save current drawing color */
   setcolor(3);      /* set drawing color to color stored in palette entry #3 */
   circle(100, 100, 5);                             /* small colored circle */
   setcolor(svcolor);                         /* restore old drawing color */

   closegraph();
};
```

getfillpattern *graphics*

Name **getfillpattern** – copies a user-defined fill pattern into memory

Usage #include <graphics.h>
void far getfillpattern(char far **upattern*);

Related functions usage void far setfillpattern(char far **upattern*, int *color*);

Prototype in graphics.h

Description **getfillpattern** copies the user-defined fill pattern, as set by **setfillpattern**, into the 8-byte area pointed to by *upattern*.

setfillpattern is like **setfillstyle**, except that you use it to set a user-defined 8x8 pattern rather than a predefined pattern.

upattern is a pointer to a sequence of 8 bytes, with each byte corresponding to 8 pixels in the pattern. Whenever a bit in a pattern byte is set to 1, the corresponding pixel

will be plotted. For example, the following user-defined fill pattern represents a checkerboard:

```
char checkboard[8] = {
   0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55
};
```

Return value None

Portability Similar routines exist in Turbo Pascal 4.0.

See also **getfillsettings**

getfillsettings *graphics*

Name **getfillsettings** – gets information about current fill pattern and color

Usage #include <graphics.h>
void far getfillsettings(struct fillsettingstype far **fillinfo*);

Related functions usage void far setfillstyle(int *pattern*, int *color*);

Prototype in graphics.h

Description The functions **bar, bar3d, fillpoly, floodfill,** and **pieslice** all fill an area with the current fill pattern in the current fill color. There are 11 predefined fill pattern styles (such as solid, cross-hatch, dotted, etc.). Symbolic names for the predefined patterns are provided by the enumeration *fill_patterns* in GRAPHICS.H (see the following table). In addition, you can define your own fill pattern.

getfillsettings fills in the **fillsettingstype** structure pointed to by *fillinfo* with information about the current fill pattern and fill color. The **fillsettingstype** structure is defined in GRAPHICS.H as follows:

```
struct fillsettingstype {
   int pattern;                /* current fill pattern */
   int color;                    /* current fill color */
};
```

If *pattern* = 12 (USER_FILL), then a user-defined fill pattern is being used; otherwise, *pattern* gives the number of a predefined pattern.

setfillstyle sets the current fill pattern and fill color. To set a user-defined fill pattern,you should *not* give a *pattern* of 12 (USER_FILL) to setfillstyle; instead, call **setfillpattern.**

The enumeration *fill_patterns*, defined in GRAPHICS.H, gives names for the predefined fill patterns, plus an indicator for a user-defined pattern:

Name	*Value*	*Description*
EMPTY_FILL	0	Fill with background color
SOLID_FILL	1	Solid fill
LINE_FILL	2	Fill with ——
LTSLASH_FILL	3	Fill with ///
SLASH_FILL	4	Fill with ///, thick lines
BKSLASH_FILL	5	Fill with \\\, thick lines
LTBKSLASH_FILL	6	Fill with \\\
HATCH_FILL	7	Light hatch fill
XHATCH_FILL	8	Heavy cross hatch fill
INTERLEAVE_FILL	9	Interleaving line fill
WIDEDOT_FILL	10	Widely spaced dot fill
CLOSEDOT_FILL	11	Closely spaced dot fill
USER_FILL	12	User-defined fill pattern

All but EMPTY_FILL fill with the current fill color; EMPTY_FILL uses the current background color.

Return value None

If invalid input is passed to **setfillstyle, graphresult** will return –11 and the current fill pattern and fill color will remain unchanged.

Portability Similar routines exist in Turbo Pascal 4.0

See also **arc, bar, fillpoly, floodfill, getfillpattern**

Example

```
#include <graphics.h>

main()
{
   int graphdriver = DETECT, graphmode;        /* will request autodetection */

   struct fillsettingstype save;
   char savepattern[8];
   char gray50[] = { 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55 };
```

```
    initgraph(&graphdriver, &graphmode, "");            /* initialize graphics */

    getfillsettings(&save);                          /* retrieve current settings */
    if (save.pattern == USER_FILL)                     /* if user-defined pattern */
       getfillpattern(savepattern);               /* then save user fill pattern */

    setfillstyle(SLASH_FILL, BLUE);                         /* change fill style */
    bar( 0,   0, 100, 100);                      /* draw slash-filled blue bar */

    setfillpattern(gray50, YELLOW);                       /* custom fill pattern */
    bar(100, 100, 200, 200);                     /* draw customized yellow bar */

    if (save.pattern == USER_FILL)                     /* if user-defined pattern */
       setfillpattern(savepattern, save.color);       /* then restore user fill
                                                                      pattern */
    else
       setfillstyle(save.pattern, save.color);                /* restore old style */

    closegraph();
};
```

getgraphmode *graphics*

Name **getgraphmode** – returns the current graphics mode.

Usage #include <graphics.h>
int far getgraphmode(void);

Related functions usage void far setgraphmode(int *mode*);

Prototype in graphics.h

Description **getgraphmode** returns the current graphics mode set by **initgraph** or **setgraphmode.**

setgraphmode selects a graphics mode different than the default one set by **initgraph.** *mode* must be a valid mode for the current device driver. **setgraphmode** clears the screen and resets all graphics settings to their defaults (CP, palette, color, viewport, and so on). You can use **setgraphmode** in conjunction with **restorecrtmode** to switch back and forth between text and graphics modes.

Your program must make a successful call to **initgraph** before calling either of these functions.

The enumeration *graphics_mode,* defined in GRAPHICS.H, gives names for the predefined graphics modes. For a table listing these enumeration values, refer to the description for **initgraph.**

Return value None

If you give **setgraphmode** an invalid mode for the current device driver, **graphresult** will return a value of –10.

Portability Similar routines exist in Turbo Pascal 4.0

See also **getmoderange, initgraph, restorecrtmode**

Example

```
int cmode;

cmode = getgraphmode();                                    /* save current mode */
restorecrtmode();                                            /* switch to text */
printf("Now in text mode - press any key to go back to graphics ...");
getch();
setgraphmode(cmode);                                      /* back to graphics */
```

getimage *graphics*

Name **getimage** – saves a bit image of the specified region into memory.

Usage #include <graphics.h>
void far getimage(int *left*, int *top*, int *right*, int *bottom*, void far **bitmap*);

Related functions usage unsigned far imagesize(int *left*, int *top*, int *right*, int *bottom*);

void far putimage(int *left*, int *top*, void far **bitmap*, int *op*);

Prototype in graphics.h

Description These three functions are used for copying an image from the screen to memory, then putting it back on the screen.

getimage saves a bit image of a rectangular region on the screen to memory. *left, top, right,* and *bottom* define the on-screen rectangle. *bitmap* points to the area in memory where the bit image will be stored. The first two words of this area are used for the width and height of the rectangle; the remainder holds the image itself.

imagesize determines the number of bytes necessary for **getimage** to save the specified rectangle. The image size returned includes space for the width and height of the rectangle.

putimage puts the bit image previously saved with **getimage** back onto the screen, with the upper-left corner of the image placed at (*left,top*). *bitmap* points to the area in memory where the source image is stored.

The *op* parameter to **putimage** specifies a combination operator that controls how the color for each destination pixel on screen is computed, based on the pixel already on screen and the corresponding source pixel in memory.

The enumeration *putimage_ops*, defined in GRAPHICS.H, gives names to these operators:

Name	*Value*	*Description*
COPY_PUT	0	copy
XOR_PUT	1	exclusive-or
OR_PUT	2	inclusive-or
AND_PUT	3	and
NOT_PUT	4	copy the inverse of the source

In words, COPY_PUT will copy the source bitmap image onto the screen, XOR_PUT will XOR the source image with that already on screen, OR_PUT will OR the source image with that on screen, etc.

Return value **imagesize** returns the size of the required memory area. If the size required for the selected image is greater than or equal to 64K bytes, **imagesize** returns 0xFFFF (-1).

getimage and **putimage** return nothing.

Portability Similar routines exist in Turbo Pascal 4.0

See also

Example

```
#include <graphics.h>

main()
{
```

```
    int graphdriver = DETECT, graphmode;       /* will request autodetection */

    void * buffer;
    unsigned size;

    initgraph(&graphdriver, &graphmode, "");           /* initialize graphics */

    size=imagesize(0,0,20,10);
    buffer=malloc (size);                              /*get memory for image */
    getimage(0,0,20,10,buffer);                                 /*save bits */

    /*   ...   */

    putimage(0,0,buffer,COPY_PUT);                           /*restore bits */
    free(buffer);                                             /*free buffer*/

    closegraph();
}
```

getlinesettings *graphics*

Name **getlinesettings** – gets the current line style, pattern, and thickness

Usage #include <graphics.h>
void far getlinesettings(struct linesettingstype far **lineinfo*);

Related functions usage void far setlinestyle(int *linestyle*, unsigned *upattern*, int *thickness*);

Prototype in graphics.h

Description **getlinesettings** fills a **linesettingstype** structure pointed to by *lineinfo* with information about the current line style, pattern, and thickness.

You can change these values by calling **setlinestyle**; this function sets the style for all lines drawn by **line, lineto, rectangle, drawpoly, arc, circle, ellipse, pieslice,** etc.

The **linesettingstype** structure is defined in GRAPHICS.H as follows:

```
struct linesettingstype {
   int linestyle;
   unsigned upattern;
   int thickness;
};
```

linestyle specifies in which of several styles subsequent lines will be drawn (such as solid, dotted, centered,

dashed). The enumeration *line_styles*, defined in GRAPHICS.H, gives names to these operators:

Name	*Value*	*Description*
SOLID_LINE	0	Solid line
DOTTED_LINE	1	Dotted line
CENTER_LINE	2	Centered line
DASHED_LINE	3	Dashed line
USERBIT_LINE	4	User-defined line style

thickness specifies whether the width of subsequent lines drawn will be normal or thick.

Name	*Value*	*Description*
NORM_WIDTH	1	1 pixel wide
THICK_WIDTH	3	3 pixels wide

upattern is a 16-bit pattern that applies only if *linestyle* is USERBIT_LINE (4). In that case, whenever a bit in the pattern word is 1, the corresponding pixel in the line is drawn in the current drawing color. For example, a solid line corresponds to a *upattern* of 0xFFFF (all pixels drawn), while a dashed line can correspond to a *upattern* of 0x3333 or 0x0F0F. If the *linestyle* parameter to **setlinestyle** is not USERBIT_LINE (!= 4), the *upattern* parameter must still be supplied but it is ignored.

Return value None

If invalid input is passed to **setlinestyle**, **graphresult** will return –11 and the current line style remains unchanged.

Portability Similar routines exist in Turbo Pascal 4.0

Example

```
#include <graphics.h>

main()
{
   int graphdriver = DETECT, graphmode;        /* will request autodetection */
   struct linesettingstype saveline;

   initgraph(&graphdriver, &graphmode, "");           /* initialize graphics */
```

```
    getlinesettings(&saveline);                    /* save current line style */
    setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
    rectangle(10, 10, 17, 15);                     /* draw a little thick box */
    setlinestyle(saveline.linestyle,              /* restore old line settings */
                 saveline.pattern,
                 saveline.thickness);

    closegraph();
};
```

getmaxcolor *graphics*

Name **getmaxcolor** – returns maximum color value

Usage #include <graphics.h>
int far getmaxcolor(void);

Prototype in graphics.h

Description **getmaxcolor** returns the highest valid pixel value (palette *size* – 1) for the current graphics driver and mode.

Return value **getmaxcolor** returns the highest available color.

Portability A similar routine exists in Turbo Pascal 4.0

See also **getbkcolor, getpalette**

getmaxx *graphics*

Name **getmaxx** – returns maximum *x* screen coordinate

Usage #include <graphics.h>
int far getmaxx(void);

Related functions usage int far getmaxy(void);

Prototype in graphics.h

Description **getmaxx** returns the maximum (screen-relative) *x* value for the current graphics driver and mode.

getmaxy returns the maximum (screen-relative) *y* value for the current graphics driver and mode.

For example, on a CGA in 320×200 mode, **getmaxx** returns 319, and **getmaxy** returns 199. **getmaxx** and

	getmaxy are invaluable for centering, determining the boundaries of a region on the screen, and so on.
Return value	**getmaxx** returns the maximum *x* screen coordinate; **getmaxy** returns the maximum *y* screen coordinate.
Portability	Similar routines exist in Turbo Pascal 4.0
See also	**getx**
Example	

```
printf("The screen resolution is %d pixels by %d pixels.\n",
       getmaxx()+1, getmaxy()+1);
```

getmaxy *graphics*

Name	**getmaxy** – returns maximum *y* screen coordinate
Usage	#include <graphics.h> int far getmaxy(void);
Prototype in	graphics.h
Description	see **getmaxx**

getmoderange *graphics*

Name	**getmoderange** – gets the range of modes for a given graphics driver
Usage	#include <graphics.h> void far getmoderange(int *graphdriver*, int far **lomode*, int far **himode*); Prototype in graphics.h
Description	**getmoderange** gets the range of valid graphics modes for the given graphics driver, *graphdriver*. The lowest permissible mode value is returned in **lomode* and the highest value is returned in **himode*. If *graphdriver* specifies an invalid graphics driver, both **lomode* and **himode* are set to –1.
Return value	None
See also	**initgraph, getgraphmode**

Example

```
int lo, hi;

getmoderange(CGA, &lo, &hi);
printf("CGA supports modes %d through %d\n", lo, hi);
```

getpalette *graphics*

Name **getpalette** – returns information about the current palette

Usage #include <graphics.h>
void far getpalette(struct palettetype far **palette*);

Related functions usage void far setallpalette(struct palettetype far **palette*);
void far setpalette(int *index*, int *actual_color*);

Prototype in graphics.h

Description **getpalette** fills the **palettetype** structure pointed to by *palette* with information about the current palette's size and colors.

You can partially (or completely) change the colors in the EGA/VGA palette with **setpalette** (or **setallpalette**). On a CGA, you can only change the first entry in the palette (*index* = 0, the background color) with a call to **setpalette**.

setallpalette sets the current palette to the values given in the **palettetype** structure pointed to by *palette*.

setpalette changes the *index* entry in the palette to *actual_color*. For example, `setpalette(0, 5)` changes the first color in the current palette (the background color) to actual color number 5. If *size* is the number of entries in the current palette, *index* can range between 0 and (*size*-1).

The **palettetype** structure (used by **getpalette** and **setallpalette**) and the MAXCOLORS constant are defined in GRAPHICS.H as follows:

```
#define MAXCOLORS  15

struct palettetype {
   unsigned char size;
   signed char colors[MAXCOLORS + 1];
};
```

size gives the number of colors in the palette for the current graphics driver in the current mode.

colors is an array of *size* bytes containing the actual raw color numbers for each entry in the palette. In the **setallpalette** routine, if an element of *colors* is –1, the palette color for that entry is not changed.

The *actual_color* parameter passed to **setpalette,** as well as the elements in the *colors* array used by **setallpalette,** can be represented by symbolic constants defined in GRAPHICS.H.

ACTUAL COLOR TABLE

CGA		*EGA/VGA*	
Name	Value	*Name*	Value
BLACK	0	EGA_BLACK	0
BLUE	1	EGA_BLUE	1
GREEN	2	EGA_GREEN	2
CYAN	3	EGA_CYAN	3
RED	4	EGA_RED	4
MAGENTA	5	EGA_MAGENTA	5
BROWN	6	EGA_BROWN	20
LIGHTGRAY	7	EGA_LIGHTGRAY	7
DARKGRAY	8	EGA_DARKGRAY	56
LIGHTBLUE	9	EGA_LIGHTBLUE	57
LIGHTGREEN	10	EGA_LIGHTGREEN	58
LIGHTCYAN	11	EGA_LIGHTCYAN	59
LIGHTRED	12	EGA_LIGHTRED	60
LIGHTMAGENTA	13	EGA_LIGHTMAGENTA	61
YELLOW	14	EGA_YELLOW	62
WHITE	15	EGA_WHITE	63

Note that valid colors depend on the current graphics driver and current graphics mode.

Changes made to the palette are seen immediately on the screen. Each time a palette color is changed, all occurrences of that color on the screen will change to the new color value.

Return value None

If invalid input is passed to **setpalette, graphresult** will return –11 and the current palette remains unchanged.

Portability A similar routine exists in Turbo Pascal 4.0

See also getbkcolor, getcolor

Example

```
#include <graphics.h>

main()
{
   int graphdriver = DETECT, graphmode;        /* will request autodetection */
   struct palettetype palette;
   int color;

   initgraph(&graphdriver, &graphmode, "");            /* initialize graphics */

   getpalette(&palette);                               /* get current palette */
   for(color=0; color<palette.size; color++)
   {
      setfillstyle(SOLID_FILL, color);             /* draw some colorful bars */
      bar(20*(color-1), 0, 20*color, 20);
   };

   if (palette.size > 1)                         /* only if more than 1 color */
   {
      do                                            /* switch colors randomly */
         setpalette(random(palette.size), random(palette.size));
      while(!kbhit());                                 /* until a key is hit */
      getch();                                          /* discard keystroke */
   };

   setallpalette(&palette);                      /* restore original palette */

   closegraph();
};
```

getpixel *graphics*

Name **getpixel** – gets the color of a specified pixel

Usage #include <graphics.h>
int far getpixel(int *x*, int *y*);

Related functions usage void far putpixel(int *x*, int *y*, int *pixelcolor*);

Prototype in graphics.h

Description **getpixel** gets the color of the pixel located at (*x*,*y*).

putpixel plots a point in the color defined by *pixelcolor* at (*x*,*y*).

Return value getpixel returns the color of the given pixel; **putpixel** returns nothing.

Portability Similar routines exist in Turbo Pascal 4.0

See also **getimage**

Example

```
#include <graphics.h>

main()
{
   int graphdriver = DETECT, graphmode;       /* will request autodetection */
   int i, color, max;

   initgraph(&graphdriver, &graphmode, "");          /* initialize graphics */
   max = getmaxcolor() + 1;

   /* change color of pixels in a diagonal line */
   for (i=1; i<200; i++) {
      color = getpixel(i,i);
      putpixel(i, i, (color ^ i) % max);
   }
   closegraph();
}
```

gettext *text*

Name **gettext** – copies text from text-mode screen to memory

Usage int gettext(int *left*, int *top*, int *right*, int *bottom*, void **destin*);

Related functions usage int puttext(int *left*, int *top*, int *right*, int *bottom*, void **source*,);

Prototype in conio.h

Description **gettext** stores the contents of an on-screen rectangle defined by *left*, *top*, *right*, and *bottom*, into the area of memory pointed to by *destin*.

puttext writes the contents of the memory area pointed to by *source* out to the on-screen rectangle defined by *left*, *top*, *right*, and *bottom*.

All coordinates are absolute screen coordinates, not window-relative.

gettext reads the contents of the rectangle into memory sequentially from left to right and top to bottom. **puttext** places the contents of a memory into the defined rectangle in the same manner.

Each position on screen takes 2 bytes of memory: The first byte is the character in the cell, and the second is the cell's video attribute. The space required for a rectangle *w* columns wide by *h* rows high is defined as:

$$bytes = (h \text{ rows}) \times (w \text{ columns}) \times 2$$

Return value These functions return 1 if the operation succeeded; they return 0 if it failed (for example, if you gave coordinates outside the range of the current screen mode).

Portability These text mode functions work on IBM PCs and BIOS-compatible systems, only.

See also **movetext**

Example

```
char buf[20*10*2];
gettext(0,0,20,10,buf);                             /* save rectangle */
/* ... */
puttext(0,0,buf);                                   /* restore screen */
```

gettextinfo *text*

Name **gettextinfo** – gets text mode video information

Usage #include <conio.h>
void gettextinfo(struct text_info **inforec*);

Prototype in conio.h

Description **gettextinfo** fills in the **text_info** structure pointed to by *inforec* with the current text video information.

The **text_info** structure is defined in CONIO.H as follows:

```
struct text_info {
   unsigned char winleft;          /* left window coordinate */
   unsigned char wintop;            /* top window coordinate */
   unsigned char winright;        /* right window coordinate */
   unsigned char winbottom;      /* bottom window coordinate */
```

```
    unsigned char attribute;              /* text attribute */
    unsigned char normattr;             /* normal attribute */
    unsigned char currmode;       /* BW40, BW80, C40, or C80 */
    unsigned char screenheight;            /* bottom - top */
    unsigned char screenwidth;             /* right - left */
    unsigned char curx;     /* x coordinate in current window*/
    unsigned char cury;     /* y coordinate in current window*/
};
```

Return value None. The results are returned in the structure pointed to by *inforec*.

Portability This function works with IBM PCs and compatibles, only.

See also **textattr, textbackground, textcolor, textmode, wherex, wherey, window**

Example

```
#include <conio.h>
struct text_info initial_info;
main()
{
   gettextinfo(&initial_info);
   /* ... */
   /* restore text mode to original value */
   textmode(initial_info.currmode);
}
```

gettextsettings *graphics*

Name **gettextsettings** – returns information about the current text settings

Usage #include <graphics.h>
void far gettextsettings(struct textsettingstype
far **textinfo*);

Related functions usage void far settextjustify(int *horiz*, int *vert*);
void far settextstyle(int *font*, int *direction*, int *charsize*);

Prototype in graphics.h

Description **gettextsettings** fills the **textsettingstype** structure pointed to by *textinfo* with information about the current

text font, direction, size, and justification (as set by **settextstyle** and **settextjustify**).

The **textsettingstype** structure used by **gettextsettings** is defined in GRAPHICS.H as follows:

```
struct textsettingstype {
   int font;
   int direction;
   int charsize;
   int horiz;
   int vert;
};
```

Text output after a call to **settextjustify** will be justified around the CP horizontally and vertically as specified. The default justification settings are LEFT_TEXT (for horizontal) and TOP_TEXT (for vertical). The enumeration *text_just* in GRAPHICS.H provides names for the *horiz* and *vert* settings passed to **settextjustify**:

Name	*Value*	*Description*
LEFT_TEXT	0	horiz
CENTER_TEXT	1	horiz and vert
RIGHT_TEXT	2	horiz
BOTTOM_TEXT	0	vert
TOP_TEXT	2	vert

If *horiz* is equal to *LEFT_TEXT* and *direction* = HORIZ_DIR, the CP's *x* component (CPX) is advanced after a call to `outtext(string)` by `textwidth(string)`.

settextstyle sets the text font, the direction in which text is displayed, and the size of the characters. A call to **settextstyle** affects all text output by **outtext** and **outtextxy**.

The parameters *font, direction,* and *charsize* passed to **settextstyle** are described in the following:

font: one 8×8 bit-mapped font and several "stroked" fonts are available. The 8×8 bit-mapped font is the default. The enumeration *font_names*, defined in GRAPHICS.H, provides names for these different font settings (see following table).

Name	*Value*	*Description*
DEFAULT_FONT	0	8x8 bit-mapped font
TRIPLEX_FONT	1	Stroked triplex font
SMALL_FONT	2	Stroked small font
SANSSERIF_FONT	3	Stroked sans-serif font
GOTHIC_FONT	4	Stroked gothic font

The default bit-mapped font is built into the graphics system. Stroked fonts are stored in *.CHR disk files, and only one at a time is kept in memory. Therefore, when you select a stroked font (different from the last selected stroked font), the corresponding *.CHR file must be loaded from disk. To avoid this loading when several stroked fonts are used, you can link font files into your program. You do this by converting them into object files with the BGIOBJ utility, then registering them through **registerbgifont**, as described in Appendix D of this addendum.

direction: font directions supported are horizontal text (left to right) and vertical text (rotated 90 degrees counterclockwise). The default direction is HORIZ_DIR.

Name	*Value*	*Description*
HORIZ_DIR	0	left to right
VERT_DIR	1	bottom to top

charsize: the size of each character can be magnified using the *charsize* factor. If *charsize* is non-zero, it can affect bit-mapped or stroked characters. If *charsize* = 0, only stroked characters are affected.

- If *charsize* = 1, **outtext** and **outtextxy** will display characters from the 8×8 bit-mapped font in an 8×8 pixel rectangle on the screen.
- If *charsize* = 2, these output functions will display characters from the 8×8 bit-mapped font in a 16×16 pixel rectangle; and so on (up to a limit of 10 times the normal size).

- When *charsize* = 0, the output functions **outtext** and **outtextxy** magnify the stroked font text using either the default character magnification factor (4), or the user-defined character size given by **setusercharsize**.

Always use **textheight** and **textwidth** to determine the actual dimensions of the text.

Return value None

Since stroked fonts can be stored on disk, errors can occur when trying to load them. If an error occurs, **graphresult** will return one of the following values:

–8	Font file not found.
–9	Not enough memory to load the font selected.
–11	(general error)
–12	Graphics I/O error
–13	Invalid font file
–14	Invalid font number

If invalid input is passed to **settextjustify**, **graphresult** will return –11 and the current text justification remains unchanged.

Portability Similar routines exist in Turbo Pascal 4.0

See also **graphresult, outtext, registerbgifont, textheight**

Example

```
#include <graphics.h>

main()
{
   int graphdriver = DETECT, graphmode;         /* will request autodetection */
   struct textsettingstype oldtext;


   initgraph(&graphdriver, &graphmode, "");          /* initialize graphics */

   gettextsettings(&oldtext);                       /* get current settings */

   /* switch to horizontal, upper-left-justified, */
   /* gothic font, scaled by a factor of 5 */

   settextjustify(LEFT_TEXT, TOP_TEXT);
   settextstyle(GOTHIC_FONT, HORIZ_DIR, 5);
   outtext("Gothic Text");

   /* restore previous settings */
   settextjustify(oldtext.horiz, oldtext.vert);
   settextstyle(oldtext.font, oldtext.direction, oldtext.charsize);
```

```
    closegraph();
}
```

getviewsettings *graphics*

Name **getviewsettings** – returns information about the current viewport

Usage #include <graphics.h>
void far getviewsettings(struct viewporttype
far **viewport*);

Related functions usage void far setviewport(int *left*, int *top*, int *right*, int *bottom*,
int *clipflag*);

Prototype in graphics.h

Description **getviewsettings** fills the **viewporttype** structure pointed to by *viewport* with information about the current viewport.

setviewport establishes a new viewport for graphics output.

The viewport's corners are given in absolute screen coordinates by (*left,top*) and (*right,bottom*). The current position (CP) is moved to viewport (0,0).

The parameter *clipflag* determines whether drawings are clipped (truncated) at the current viewport boundaries. If *clipflag* is non-zero when your program calls **setviewport**, all drawings will be clipped to the current viewport.

The **viewporttype** structure used by **getviewport** is defined in GRAPHICS.H as follows:

```
struct viewpcrttype {
   int left, top, right, bottom;
   int clipflag;
};
```

Note: **initgraph** and **setgraphmode** reset the viewport to the entire graphics screen.

Return value None

If invalid input is passed to **setviewport**, **graphresult** will return –11 and the current view settings remain unchanged.

Portability A similar routine exists in Turbo Pascal 4.0

See also **clearviewport**

Example

```
struct viewporttype view;

getviewsettings(&view);                              /* get current setting */
if (!view.clip)                                       /* if clipping not on */
   setviewport(view.left,  view.top,
               view.right, view.bottom, 1);                  /* turn it on */
```

getx *graphics*

Name **getx** – returns the current position's *x* coordinate

Usage #include <graphics.h>
int far getx(void);

Related functions usage int far gety(void);

Prototype in graphics.h

Description **getx** returns the current position's *x* coordinate.

gety returns the current position's *y* coordinate.

Return value **getx** returns the CP's *x* coordinate; **gety** returns the CP's *y* coordinate. (The values are viewport-relative.)

Portability Similar routines exist in Turbo Pascal 4.0

See also **getviewsettings, initgraph, moveto**

Example

```
int oldx, oldy;

/* save current position */
oldx = getx();
oldy = gety();

circle(100, 100, 2);                        /* draw a blob at [100,100] */
moveto(99,100);
linerel(2,0);
moveto(oldx, oldy);                         /* back to the old position */
```

gety *graphics*

Name gety – returns the current position's *y* coordinate

Usage #include <graphics.h>
int far gety(void);

Prototype in graphics.h

Description see **getx**

gotoxy *text*

Name **gotoxy** – positions cursor in text window

Usage void gotoxy(int *x*, int *y*);

Prototype in conio.h

Description **gotoxy** moves the cursor to the given position in the current text window. If the coordinates are in any way invalid, the call to **gotoxy** is ignored. An example of this is a call to `gotoxy(40,30)` when (35,25) is the bottom right position in the window.

The two functions **wherex** and **wherey** will return the current *x* and *y* positions of the cursor, respectively.

Return value None

Portability This function works with IBM PCs and compatibles, only; a corresponding function exists in Turbo Pascal.

See also **wherex, wherey, window**

Example

```
gotoxy(10,20);                    /* position cursor at column 10, row 20 */
```

graphdefaults *graphics*

Name **graphdefaults** – resets all graphics settings to their defaults

Usage #include <graphics.h>
void far graphdefaults(void);

Prototype in graphics.h

Description **graphdefaults** resets all graphics settings to their defaults. It:

- sets the viewport to the entire screen
- moves the current position to (0,0)
- sets the default palette colors, background color, and drawing color
- sets the default fill style and pattern
- sets the default text font and justification

Return value None

Portability A similar routine exists in Turbo Pascal 4.0

See also **initgraph, getgraphmode**

grapherrormsg *graphics*

Name grapherrormsg – returns an error message string

Usage #include <graphics.h>
char far *far grapherrormsg(int *errorcode*);

Prototype in graphics.h

Description see **graphresult**

_graphfreemem *graphics*

Name **_graphfreemem** – user-modifiable graph memory deallocation

Usage #include <graphics.h>
void far _graphfreemem(void far **ptr*, unsigned *size*);

Prototype in graphics.h

Description see **_graphgetmem**

_graphgetmem *graphics*

Name — **_graphgetmem** – user-modifiable graphics memory allocation

Usage — #include <graphics.h>
void far * far _graphgetmem(unsigned *size*);

Related functions usage — void far _graphfreemem(void far **ptr*, unsigned *size*);

Prototype in — graphics.h

Description — The graphics library calls **_graphgetmem** to allocate memory for internal buffers, graphics drivers, and character sets. You can choose to control the memory management of the graphics library by defining your own version of **_graphgetmem** (you must declare it exactly as shown in the Usage). The default version of this routine merely calls **malloc**.

The graphics library calls **_graphfreemem** to release memory previously allocated through **_graphgetmem**. You can choose to control the graphics library memory management by defining your own version of **_graphfreemem** (you must declare it exactly as shown in the Usage). The default version of this routine merely calls **free**.

Return value — None

Portability — Similar routines exist in Turbo Pascal 4.0.

Example

```
/* example of user-defined graph management routines */

#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <alloc.h>

main()
{
   int errorcode;
   int graphdriver;
   int graphmode;

   graphdriver = DETECT;
```

```
    initgraph(&graphdriver, &graphmode, "c:\\");
    errorcode = graphresult();
    if (errorcode != grOk)
    {
       printf("graphics error: %s\n",grapherrormsg(errorcode));
       exit(1);
    };

    settextstyle(GOTHIC_FONT, HORIZ_DIR, 4);
    outtextxy( 100, 100, "BGI TEST");

    closegraph();
}

void far * far _graphgetmem(unsigned size)
{
    printf("_graphgetmem  called [size=%d] -- hit any key",size);
    getch(); printf("\n");
    return(farmalloc(size));                              /* use "far" heap */
}

void far _graphfreemem(void far *ptr, unsigned size)
{
    printf("_graphfreemem called [size=%d] -- hit any key",size);
    getch(); printf("\n");
    farfree(ptr);                                         /* "size" not used */
}
```

graphresult *graphics*

Name **graphresult** – returns an error code for the last unsuccessful graphics operation

Usage #include <graphics.h>
int far graphresult(void);

Related functions usage char far *far grapherrormsg(int *errorcode*);

Prototype in graphics.h

Description **graphresult** returns the error code for the last graphics operation that reported an error.

grapherrormsg returns a pointer to a string associated with *errorcode,* the error code returned by **graphresult.** This makes it easy for your program to display a descriptive error message, such as "Device driver not found (CGA.BGI)" instead of "error code -3", which makes your programs easier for other humans to follow.

The following table lists the error codes returned by **graphresult**, the *graphics_errors* type constant associated with the error codes, and the corresponding error messages:

error code	*graphics_errors* constant	corresponding error message string
0	grOk	No error
–1	grNoInitGraph	(BGI) graphics not installed (use **initgraph**)
–2	grNotDetected	Graphics hardware not detected
–3	grFileNotFound	Device driver file not found
–4	grInvalidDriver	Invalid device driver file
–5	grNoLoadMem	Not enough memory to load driver
–6	grNoScanMem	Out of memory in scan fill
–7	grNoFloodMem	Out of memory in flood fill
–8	grFontNotFound	Font file not found
–9	grNoFontMem	Not enough memory to load font
–10	grInvalidMode	Invalid graphics mode for selected driver
–11	grError	Graphics error
–12	grIOerror	Graphics I/O error
–13	grInvalidFont	Invalid font file
–14	grInvalidFontNum	Invalid font number
–15	grInvalidDeviceNum	Invalid device number

Note that **graphresult** is reset to 0 after it has been called. Therefore, you should store the value of **graphresult** into a temporary variable and then test it.

Return value **graphresult** will return the current graphics error number, an integer in the range –15 to 0; **grapherrormsg** returns a pointer to a string associated with the value returned by **graphresult**.

Portability A similar routine exists in Turbo Pascal 4.0

See also **initgraph**

highvideo *text*

Name **highvideo** – selects high intensity text characters

Usage void highvideo(void);

Related functions usage void lowvideo(void);
void normvideo(void);

Prototype in conio.h

Description **highvideo** selects high intensity characters by setting the high intensity bit of the currently selected foreground color.

normvideo selects normal characters by returning the text attribute (foreground and background) to the value it had when the program started.

lowvideo selects low intensity characters by clearing the high intensity bit of the currently selected foreground color.

These functions do not affect any characters currently on the screen; they only affect those displayed using direct console output functions (such as **cprintf**) *after* these functions are called.

Return value None

Portability These functions work with IBM PCs and compatibles, only; corresponding functions exist in Turbo Pascal.

See also **cprintf, cputs, gettextinfo, putch, textattr**

imagesize *graphics*

Name **imagesize** – returns the number of bytes required to store a bit image

Usage #include <graphics.h>
unsigned far imagesize(int *left*, int *top*,
int *right*, int *bottom*);

Prototype in graphics.h

Description see **getimage**

Name	**initgraph** – initializes the graphics system
Usage	#include <graphics.h> void far initgraph(int far **graphdriver*, int far **graphmode*, char far **pathtodriver*);
Related functions usage	void far detectgraph(int far **graphdriver*, int far **graphmode*); void far closegraph(void);
Prototype in	graphics.h

Description

initgraph initializes the graphics system by loading a graphics driver from disk (or validating a registered driver), and putting the system into graphics mode.

detectgraph detects your system's graphics adapter and chooses the mode that provides the highest resolution for that adapter. If no graphics hardware was detected, the **graphdriver* parameter is set to –2 and **graphresult** will also return –2.

Note: The main reason to call **detectgraph** directly is to override the graphics mode that **detectgraph** recommends to **initgraph.**

closegraph deallocates all memory allocated by the graphics system, then restores the screen to the mode it was in before you called **initgraph.** (The graphics system deallocates memory, such as the drivers, fonts, and an internal buffer, through a call to **_graphfreemem.**)

To start the graphics system, you first call the **initgraph** function. **initgraph** loads the graphics driver and puts the system into graphics mode. You can tell **initgraph** to use a particular graphics driver and mode, or to auto detect the attached video adapter at run time and pick the corresponding driver. If you tell **initgraph** to auto detect, it calls **detectgraph** to select a graphics driver and mode. **initgraph** also resets all graphics settings to their defaults (current position, palette, color, viewport, etc.) and resets **graphresult** to 0.

Normally, **initgraph** loads a graphics driver by allocating memory for the driver (through **_graphgetmem**), then loading the appropriate .BGI file from disk. As an alternative to this dynamic loading scheme, you can link a graphics driver file (or several of them) directly into your executable program file. *See Appendix D in this addendum for more information.*

pathtodriver specifies the directory path where **initgraph** will look for the graphics drivers. **initgraph** first looks in the path specified in *pathtodriver*, then (if they're not there) in the current directory. Accordingly, if *pathtodriver* is NULL, the driver files (*.BGI) must be in the current directory. This is also the path **settextstyle** will search for the stroked character font (*.CHR) files.

**graphdriver* is an integer that specifies the graphics driver to be used. You can give it a value using a constant of the *graphics_drivers* enumeration type, defined in GRAPHICS.H and listed in the following table.

graphics_drivers **constant**	**Numeric Value**
DETECT	0 (requests autodetection)
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
RESERVED	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

**graphmode* is an integer that specifies the initial graphics mode (unless **graphdriver* = DETECT, in which case **graphmode* is set to the highest resolution available for the detected driver). You can give **graphmode* a value using a constant of the *graphics_modes* enumeration type, defined in GRAPHICS.H and listed in the following table.

graphics driver	*graphics_modes*	Value	Column x Row	Palette	Pages
CGA	CGAC0	0	320x200	C0	1
	CGAC1	1	320x200	C1	1
	CGAC2	2	320x200	C2	1
	CGAC3	3	320x200	C3	1
	CGAHI	4	640x200	2 color	1
MCGA	MCGAC0	0	320x200	C0	1
	MCGAC1	1	320x200	C1	1
	MCGAC2	2	320x200	C2	1
	MCGAC3	3	320x200	C3	1
	MCGAMED	4	640x200	2 color	1
	MCGAHI	5	640x480	2 color	1
EGA	EGALO	0	640x200	16 color	4
	EGAHI	1	640x350	16 color	2
EGA64	EGA64LO	0	640x200	16 color	1
	EGA64HI	1	640x350	4 color	1
EGA-	EGAMONOHI	3	640x350	2 color	1*
MONO	" "	3	" "	2 color	2**
HERC	HERCMONOHI	0	720x348	2 color	2
ATT400	ATT400C0	0	320x200	C0	1
	ATT400C1	1	320x200	C1	1
	ATT400C2	2	320x200	C2	1
	ATT400C3	3	320x200	C3	1
	ATT400MED	4	640x200	2 color	1
	ATT400HI	5	640x400	2 color	1
VGA	VGALO	0	640x200	16 color	2
	VGAMED	1	640x350	16 color	2
	VGAHI	2	640x480	16 color	1
PC3270	PC3270HI	0	720x350	2 color	1

* 64K on EGAMONO card
** 256K on EGAMONO card

In the previous table, the **Palette** listings C0, C1, C2, and C3 refer to the four predefined four-color palettes available on CGA (and compatible) systems. You can select the background color (entry #0) in each of these palettes, but the other colors are fixed. These palettes are described in greater detail in Chapter 1 of this

addendum (under "Color Control") and summarized in the following table.

Palette Number	*Color assigned to pixel value* 1	2	3
0	lightgreen	lightred	yellow
1	lightcyan	lightmagenta	white
2	green	red	brown
3	cyan	magenta	lightgray

After a call to **initgraph**, **graphdriver* is set to the current graphics driver, and **graphmode* is set to the current graphics mode.

Return value None

initgraph always sets the internal error code; on success, it sets the code to 0. If an error occurred, **graphdriver* is set to –2, –3, –4, or –5, and **graphresult** returns the same value, as listed here:

–2	Cannot detect a graphics card
–3	Cannot find driver file
–4	Invalid driver
–5	Insufficient memory to load driver

Portability Similar routines exist in Turbo Pascal 4.0

See also **getgraphmode, _graphgetmem, initgraph, registerbgidriver, restorecrtmode, setgraphbufsize**

Example

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>

main()
{
   int g_driver, g_mode, g_error;

   detectgraph(&g_driver, &g_mode, "");

   if (g_driver < 0)
   {
      printf("No graphics hardware detected !\n");
      exit(1);
```

```
    }
    printf("Detected graphics driver #%d, mode #%d\n",g_driver,g_mode);
    getch();

    if (g_mode == EGAHI) g_mode = EGALO        /* override mode if EGA detected */
    initgraph(&g_driver, &g_mode);
    g_error = graphresult();

    if (g_error < 0)
    {
      printf("initgraph error: %s.\n", grapherrormsg(g_error));
      exit(1);
    }
    bar(0, 0, getmaxx()/2, getmaxy());
    getch();
    closegraph();
}
```

insline *text*

Name **insline** – inserts blank line in text window

Usage void insline(void);

Prototype in conio.h

Description **insline** inserts an empty line in the text window at the cursor position using the current text background color. All lines below the empty one move down one line and the bottom line scrolls off the bottom of the window.

Return value None

Portability This function works with IBM PCs and compatibles, only; a corresponding function exists in Turbo Pascal.

See also **clreol, delline, window**

ldiv *misc*

Name **ldiv** – divides two longs, returns quotient and remainder

Usage #include <stdlib.h>
ldiv_t ldiv(long *lnumer*, long *ldenom*):

Prototype in stdlib.h

Description see **div** in this addendum

lfind *modified*

Name **lfind** – linear search

Usage

```
#include <stdlib.h>
void  *lfind(const void *key, const void *base,
            size_t *pnelem, size_t width,
            int (*fcmp)(const void *, const void *));
```

Prototype in stdlib.h

Description see **bsearch** (in this addendum and in the *Turbo C Reference Guide*)

line *graphics*

Name **line** – draws a line between two specified points

Usage

```
#include <graphics.h>
void far line(int x0, int y0, int x1, int y1);
```

Related functions usage

```
void far lineto(int x, int y);
```

```
void far linerel(int dx, int dy);
```

Prototype in graphics.h

Description Each of these line-drawing functions draws a line in the current color, using the current line style and thickness.

line draws a line between the two points specified, (*x0*,*y0*) and (*x1*,*y1*), without updating the current position (CP).

lineto draws a line from the CP to (*x*,*y*), then moves the CP to (*x*,*y*).

linerel draws a line from the CP to a point that is a relative distance (*dx*,*dy*) from the CP. The CP is advanced by (*dx*,*dy*).

Return value None

Portability Similar routines exist in Turbo Pascal 4.0

See also **getcolor**, **getlinesettings**

linerel *graphics*

Name	**linerel** – draws a line a relative distance from the current position (CP)
Usage	#include <graphics.h> void far linerel(int *dx*, int *dy*);
Prototype in	graphics.h
Description	see **line**

lineto *graphics*

Name	**lineto** – draws a line from the CP to (*x*,*y*)
Usage	#include <graphics.h> void far lineto(int *x*, int *y*);
Prototype in	graphics.h
Description	see **line**

lowvideo *text*

Name	**lowvideo** – selects low intensity characters
Usage	void lowvideo(void);
Prototype in	conio.h
Description	see **highvideo**.

_lrotl *misc*

Name	**_lrotl** – rotates an unsigned long value to the left
Usage	unsigned long _lrotl(unsigned long *lvalue*, int *count*);
Prototype in	stdlib.h
Description	see **_rotl**

_lrotr *misc*

Name **_lrotr** – rotates an unsigned long value to the right

Usage unsigned long _lrotr(unsigned long *lvalue,* int *count*);

Prototype in stdlib.h

Description see **_rotl**

lsearch *modified*

Name **lsearch** – linear search

Usage

```
#include &lt;stdlib.h&gt;
void  *lsearch(const void *key, void *base,
               size_t *pnelem, size_t width,
               int (*fcmp)(const void *, const void *));
```

Prototype in stdlib.h

Description see **bsearch** (in this addendum and in the *Turbo C Reference Guide*)

malloc *modified*

Name **malloc** – allocates main memory

Usage

```
#include <stdlib.h>
void *malloc(size_t size);
```

Related functions usage

```
void *calloc(size_t nelem, size_t elsize);
void *realloc(void *ptr, size_t newsize);
```

Prototypes in stdlib.h and alloc.h

Description These functions have the same description as given in the *Turbo C Reference Guide.*

Return value These functions return the same values as given in the *Turbo C Reference Guide,* with the following additions:

If the argument *size* (for **malloc**), *elsize* (for **calloc**), or *newsize* (for **realloc**) == 0, these three functions return NULL.

moverel *graphics*

Name **moverel** – moves the current position (CP) a relative distance

Usage #include <graphics.h>
void far moverel(int *dx*, int *dy*);

Prototype in graphics.h

Description see **moveto**

movetext *text*

Name **movetext** – copies text on-screen from one rectangle to another

Usage int movetext(int *left*, int *top*, int *right*, int *bottom*,
int *newleft*, int *newtop*);

Prototype in conio.h

Description **movetext** copies the contents of the onscreen rectangle defined by *left*, *top*, *right*, and *bottom* to a new rectangle of the same dimensions. The new rectangle's upper left corner is position (*newleft*, *newtop*).

All coordinates are absolute screen coordinates.

Return value **movetext** returns 1 if the operation succeeded; if the operation failed (for example, if you gave coordinates outside the range of the current screen mode), **movetext** returns 0.

Portability These text mode functions can be used on IBM PCs and BIOS-compatible systems.

See also **gettext**

Example

```
/*
   copy the contents of the old rectangle, whose upper left corner
   is (5, 15) and whose lower right corner is (20, 25), to a new
   rectangle whose upper left corner is (10, 20).
*/

movetext(5, 15, 20, 25, 10, 20);
```

moveto *graphics*

Name	**moveto** – moves the CP to (*x*,*y*)
Usage	#include <graphics.h> void far moveto(int *x*, int *y*);
Related functions usage	void far moverel(int *dx*, int *dy*);
Prototype in	graphics.h
Description	Each of these "move current position" functions moves the CP to another position on screen. **moveto** moves the current position (CP) to viewport position (*x*,*y*). **moverel** moves the current position (CP) *dx* pixels in the *x* direction and *dy* pixels in the *y* direction.
Return value	None
Portability	Similar routines exist in Turbo Pascal 4.0

normvideo *text*

Name	**normvideo** – selects normal intensity characters
Usage	void normvideo(void);
Prototype in	conio.h
Description	see **highvideo**

nosound *text*

Name	**nosound** – turns PC speaker off
Usage	void nosound(void);
Prototype in	dos.h
Description	see **sound**

outtext *graphics*

Name — **outtext** – displays a string in the viewport

Usage — #include <graphics.h>
void far outtext(char far **textstring*);

Related functions usage — void far outtextxy(int *x*, int *y*, char far **textstring*);

Prototype in — graphics.h

Description — Each of these functions displays a text string in the viewport, using the current justification settings and the current font, direction, and size.

outtext outputs *textstring* at the CP. If the horizontal text justification is LEFT_TEXT and the text direction is HORIZ_DIR, the CP's *x* coordinate is advanced by `textwidth(textstring)` Otherwise, the CP remains unchanged.

outtextxy outputs *textstring* at the given position (*x*,*y*).

In order to maintain code compatibility when using several fonts, use the **textwidth** and **textheight** functions to determine the dimensions of the string.

Return value — None

Portability — Similar routines exist in Turbo Pascal 4.0

See also — **gettextsettings, textheight**

outtextxy *graphics*

Name — **outtextxy** – sends a string to the specified location

Usage — #include <graphics.h>
void far outtextxy(int *x*, int *y*, char far **textstring*);

Prototype in — graphics.h

Description — see **outtext**

pieslice *graphics*

Name **pieslice** – draws and fills in pie slice

Usage

```
#include <graphics.h>
void far pieslice(int x, int y, int stangle, int endangle,
                  int radius);
```

Prototype in graphics.h

Description see **arc**

putch *modified*

Name **putch** – puts character on screen

Usage int putch(int *ch*);

Prototype in conio.h

Description **putch** has been modified so output is written to the current text window. (See the *Turbo C Reference Guide* for further description.)

Return value **putch** returns *ch*, the character displayed.

Portability This function works with IBM PCs and compatibles only.

putimage *graphics*

Name **putimage** – puts a bit image onto the screen

Usage

```
#include <graphics.h>
void far putimage(int x, int y, void far *bitmap, int op);
```

Prototype in graphics.h

Description see **getimage**

putpixel *graphics*

Name **putpixel** – plots a pixel at a specified point

Usage #include <graphics.h>
void far putpixel(int *x*, int *y*, int *pixelcolor*);

Prototype in graphics.h

Description see **getpixel**

puttext *text*

Name **puttext** – copies text from memory to screen

Usage int puttext(int *left*, int *top*, int *right*, int *bottom*,
void *source*);
Prototype in
conio.h

Description see **gettext**

random *misc*

Name **random** – random number generator

Usage #include <stdlib.h>
int random(int *num*);

Related functions usage void randomize(void);

Prototype in stdlib.h

Description **random** returns a random number between 0 and (*num*-1). `random(num)` is a macro defined as `rand() % (num)`. Both *num* and the random number returned are integers.

randomize initializes the random number generator with a random value. Because **randomize** is implemented as a macro that calls the **time** function prototyped in TIME.H, we recommend that you also `#include <time.h>` when using this routine.

Return value **random** returns a number between 0 and (*num*-1). **randomize** does not return any value.

Portability Corresponding functions exist in Turbo Pascal.

See also **rand, seed**

Example

```
#include <stdlib.h>
#include <time.h>

main()          /* prints a random number of random numbers in the range 0-99 */
{
   int n;
   randomize();
   n = random(20) + 1;                  /* a random number between 1 and 20 */
   while (n-- >0)
      printf ("%d ", random (100));
   printf ("\n");
}
```

randomize *misc*

Name **randomize** – initializes random number generator

Usage #include <stdlib.h>
void randomize(void);

Prototype in stdlib.h

Description see **random**

read *modified*

Name **read** – reads from a file

Usage int read(int *handle,* void **buf,* unsigned *nbyte*);

Related functions usage int _read(int *handle,* void **buf,* unsigned *nbyte*);

Prototype in io.h

Description These functions have the same description as given in the *Turbo C Reference Guide,* with the following additions:

The maximum number of bytes that either of these functions can read is 65534, since 65535 (0xFFFF) is the same as –1, which is the error return indicator for these functions.

realloc *modified*

Name **realloc** – reallocates main memory

Usage #include <stdlib.h>
void *realloc(void **ptr*, size_t *newsize*);

Prototype in stdlib.h, alloc.h

Description see **malloc** (in this addendum and in the *Turbo C Reference Guide*)

rectangle *graphics*

Name **rectangle** – draws a rectangle

Usage #include <graphics.h>
void far rectangle(int *left*, int *top*, int *right*, int *bottom*);

Prototype in graphics.h

Description **rectangle** draws a rectangle in the current line style, thickness, and drawing color.

(left,top) is the upper left corner of the rectangle, and *(right, bottom)* is its lower right corner.

Return value None

Portability A similar routine exists in Turbo Pascal 4.0

See also **bar, getlinesettings, getcolor**

Example

```
int i;

for (i=0; i<10; i++)
   rectangle(20-2*i, 20-2*i, 10*(i+2), 10*(i+2));
```

registerbgidriver *graphics*

Name **registerbgidriver** – registers linked-in graphics driver code

Usage #include <graphics.h>
int registerbgidriver(void (**driver*)(void));

Related functions usage int registerbgifont(void (**font*)(void));

Prototype in graphics.h

Description Calling **registerbgidriver** informs the graphics system about the presence of a linked-in driver; similarly, calling **registerbgifont** signifies a linked-in stroked character font file. These routines check the linked-in code for the specified driver or font; if the code is valid, they register it in internal tables. Linked-in drivers and fonts are discussed in detail in Appendix D of this addendum.

By using the name of a linked-in file in a call to **registerbgidriver** or **registerbgifont**, you also tell the compiler (and linker) to link in the object file with that public name.

Return value Both routines return a negative graphics error code if the specified driver or font is invalid.

Otherwise, **registerbgidriver** returns an internal driver number, and **registerbgifont** returns the font number of the registered font.

Portability Similar routines exist in Turbo Pascal 4.0

See also **initgraph, gettextsettings**

Example

```
/* register the EGA/VGA driver */
if (registerbgidriver(EGAVGA_driver) < 0) exit(1);

/* register the gothic font */
if (registerbgifont(gothic_font) != GOTHIC_FONT) exit(1);
```

registerbgifont *graphics*

Name	**registerbgifont** – – registers linked-in stroked font code
Usage	#include <graphics.h> int registerbgifont(void (**font*)(void));
Prototype in	graphics.h
Description	see **registerbgidriver**

restorecrtmode *graphics*

Name	**restorecrtmode** –restores the screen mode to its pre-**initgraph** setting
Usage	#include <graphics.h> void far restorecrtmode(void);
Prototype in	graphics.h
Description	**restorecrtmode** restores the original video mode detected by **initgraph**. This function can be used in conjunction with **setgraphmode** to switch back and forth between text and graphics modes.
Return value	None
Portability	A similar routine exists in Turbo Pascal 4.0
See also	**initgraph, setgraphmode**

_rotl *misc*

Name	**_rotl** – rotates a value to the left
Usage	unsigned _rotl(unsigned *value*, int *count*);

Related functions usage	unsigned _rotr(unsigned *value*, int *count*);
	unsigned long _lrotl(unsigned long *lvalue*, int *count*);
	unsigned long _lrotr(unsigned long *lvalue*, int *count*);

Prototype in	stdlib.h
Description	Each of these functions rotates the given *value* to the left or right *count* bits. For **_lrotl** and **_lrotr**, *lvalue* is an

unsigned long; for **_rotl** and **_rotr**, the value rotated is an **unsigned**.

_rotl rotates *value* by *count* bits to the left
_rotr rotates *value* by *count* bits to the right
_lrotl rotates *lvalue* by *count* bits to the left
_lrotr rotates *lvalue* by *count* bits to the right

Return value Each of these functions returns the rotated value.

Example

```
#include <stdlib.h>

main()
{
   printf("rotate 0xABCD 4 bits left  = %04X\n", _rotl(0xABCD, 4));
   printf("rotate 0xABCD 4 bits right = %04X\n", _rotr(0xABCD, 4));
   printf("rotate 0x55555555 1 bit left  = %08lX\n", _lrotl(0x55555555L, 1));
   printf("rotate 0xAAAAAAAA 1 bit right = %08lX\n", _lrotr(0xAAAAAAAAL,1));
}
```

Output

```
rotate 0xABCD 4 bits left  = BCDA
rotate 0xABCD 4 bits right = DABC
rotate 0x55555555 1 bit left  = AAAAAAAA
rotate 0xAAAAAAAA 1 bit right = 55555555
```

_rotr *misc*

Name **_rotr** – rotates a value to the right

Usage unsigned _rotr(unsigned *value*, int *count*);

Prototype in stdlib.h

Description see **_lrotl**

setactivepage

Name **setactivepage** – sets active page for graphics output

Usage #include <graphics.h>
void far setactivepage(int *pagenum*);

Related functions usage void far setvisualpage(int *pagenum*);

Prototype in graphics.h

Description **setactivepage** makes *pagenum* the active graphics page. All subsequent graphics output will be directed to graphics page *pagenum*.

setvisualpage makes *pagenum* the visual graphics page.

The active graphics page may or may not be the one you see on screen, depending on how many graphics pages are available on your system. Only the EGA, VGA, and Hercules graphics cards support multiple pages.

With multiple graphics pages, your program can direct graphics output to an off-screen page, then quickly display the off-screen image by changing the visual page with a call to **setvisualpage**. This technique is especially useful for animation.

Return value None

Portability Similar routines exist in Turbo Pascal 4.0

Example

```
cleardevice();
setvisualpage(0);                         /* make page 0 (=blank) visible */
setactivepage(1);                          /* will use page 1 for output */
bar(50, 50, 150, 150);                         /* draw a bar in page 1 */
setvisualpage(1);                            /* show page 1 (with bar) */
```

setallpalette *graphics*

Name **setallpalette** – changes all palette colors as specified.

Usage #include <graphics.h>
void far setallpalette(struct palettetype far **palette*);

Prototype in graphics.h

Description see **getpalette**

setbkcolor *graphics*

Name **setbkcolor** – sets the current background color using the palette

Usage #include <graphics.h>
void far setbkcolor(int *color*);

Prototype in graphics.h

Description see **getbkcolor**

setcolor *graphics*

Name **setcolor** – sets the current drawing color using the palette

Usage #include <graphics.h>
void far setcolor(int *color*);

Prototype in graphics.h

Description see **getbkcolor**

setfillpattern *graphics*

Name **setfillpattern** – selects a user-defined fill pattern

Usage #include <graphics.h>
void far setfillpattern(char far **upattern*, int *color*);

Prototype in graphics.h

Description see **getfillpattern**

setfillstyle *graphics*

Name **setfillstyle** – sets the fill pattern and color

Usage #include <graphics.h>
void far setfillstyle(int *pattern*, int *color*);

Prototype in graphics.h

Description see **getfillsettings**

setgraphbufsize *graphics*

Name **setgraphbufsize** – changes the size of the internal graphics buffer

Usage #include <graphics.h>
unsigned far setgraphbufsize(unsigned *bufsize*);

Prototype in graphics.h

Description Some of the graphics routines (such as **floodfill**) use a memory buffer that is allocated when **initgraph** is called, and released when **closegraph** is called. The default size of this buffer, which is allocated by **_graphgetmem**, is 4096 bytes.

You might want to make this buffer smaller (to save memory space), or make it bigger (if, for example, a call to **floodfill** produces error –7: Out of flood memory). **setgraphbufsize** tells **initgraph** how much memory to allocate for this internal graphics buffer when it calls **_graphgetmem**.

You must call **setgraphbufsize** before calling **initgraph**.

Return value **setgraphbufsize** returns the previous size of the internal buffer.

Portability A similar routine exists in Turbo Pascal 4.0.

See also **closegraph, initgraph**

Example

```
int cbsize;

cbsize = setgraphbufsize(1000);                    /* get current size */
setgraphbufsize(cbsize);                              /* restore size */
printf("The graphics buffer is currently %d bytes.", cbsize);
```

setgraphmode *graphics*

Name **setgraphmode** – sets the system to graphics mode, clears the screen

Usage #include <graphics.h>
void far setgraphmode(int *mode*);

Prototype in graphics.h

Description see **getgraphmode**

setlinestyle *graphics*

Name **setlinestyle** – sets the current line width and style

Usage #include <graphics.h>
void far setlinestyle(int *linestyle*, unsigned *upattern*,
int *thickness*);

Prototype in graphics.h

Description see **getlinesettings**

setpalette *graphics*

Name **setpalette** – changes one palette color

Usage #include <graphics.h>
void far setpalette(int *index*, int *actual_color*);

Prototype in graphics.h

Description see **getpalette**

settextjustify *graphics*

Name **settextjustify** – sets text justification

Usage #include <graphics.h>
void far settextjustify(int *horiz*, int *vert*);

Prototype in graphics.h

Description see **gettextsettings**

settextstyle — *graphics*

Name **settextstyle** – sets the current text characteristics

Usage #include <graphics.h>
void far settextstyle(int *font,* int *direction,* int *charsize*);

Prototype in graphics.h

Description see **gettextsettings**

setusercharsize — *graphics*

Name **setusercharsize** – user-defined character magnification factor for stroked fonts

Usage #include <graphics.h>
void far setusercharsize(int *multx,* int *divx,*
int *multy,* int *divy*);

Prototype in graphics.h

Description **setusercharsize** gives you finer control over the size of text from stroked fonts. The values set by **setusercharsize** are active *only* if *charsize* = 0, as set by a previous call to **settextstyle.**

With **setusercharsize,** you specify factors by which the width and height are scaled. The default width is scaled by *multx* : *divx* and the default height is scaled by *multy* : *divy*. For example, to make text twice as wide and 50% taller than the default, set

```
multx = 2;  divx = 1;
multy = 3;  divy = 2;
```

Return value None

Portability A similar routine exists in Turbo Pascal 4.0.

See also **gettextsettings**

Example

```
#include <graphics.h>

main()
{
   int graphdriver = DETECT, graphmode;        /* will request autodetection */
   char *title = "TEXT in a BOX";
```

```
    initgraph(&graphdriver, &graphmode, "");          /* initialize graphics */

    /* draw a rectangle and fit a text string inside */
    settextjustify(CENTER_TEXT, CENTER_TEXT);
    setusercharsize(1,1,1,1);
    settextstyle(TRIPLEX_FONT, HORIZ_DIR, USER_CHAR_SIZE);
    setusercharsize(200, textwidth(title), 100, textheight(title));
    settextstyle(TRIPLEX_FONT, HORIZ_DIR, USER_CHAR_SIZE);
    rectangle(0, 0, 200, 100);
    outtextxy(100, 50, title);

    closegraph();
}
```

setviewport *graphics*

Name **setviewport** – sets the current viewport for graphics output

Usage #include <graphics.h>
void far setviewport(int *left*, int *top*, int *right*, int *bottom*, int *clipflag*);

Prototype in graphics.h

Description see **getviewsettings**

setvisualpage *graphics*

Name **setvisualpage** – sets the visual graphics page number

Usage #include <graphics.h>
void far setvisualpage(int *pagenum*);

Prototype in graphics.h

Description see **setactivepage**

sound *misc*

Name **sound** – turns PC speaker on at specified frequency

Usage void sound(unsigned *frequency*);

Related functions usage void nosound(void);

Prototype in dos.h

Description With a call to **sound**, you can turn the PC's speaker on at a given frequency. *frequency* specifies the frequency of the sound in Hertz. To turn the speaker off after a call to **sound**, call the function **nosound**.

Return value None

Portability These functions work with IBM PCs and compatibles, only; corresponding functions exist in Turbo Pascal.

See also **delay, sleep**

Example

```
/* emits a 7-Hz tone for 10 seconds */
/* True story: 7 Hertz is the resonant frequency of a chicken's skull cavity.
   This was determined empirically in Australia, where a new factory generating
   7-Hz tones was located too close to a chicken ranch: when the factory
   started up, all the chickens died.

   Your PC may not be able to emit a 7-Hz tone. */

   main()
   {
      sound(7);
      delay(10000);
      nosound();
   }
```

spawn... *modified*

Name **spawn...** – functions that create and run other programs

Usage Refer to *Turbo C Reference Guide*

Prototypes in process.h

Description These functions have the same description as given in the *Turbo C Reference Guide*, with the following exception:

The description (given in the *Turbo C Reference Guide*) of how **spawn...** functions search for files is not complete; the **spawn...** functions search for *pathname* as follows.

- If no explicit extension is given (for example, *pathname* = MYPROG), the functions will search for the file as given. If that one is not found, they will add .COM and search again. If that's not found, they'll add .EXE and search one last time.

- If an explicit extension or period is given (for example, *pathname* = `MYPROG.EXE`), the functions will search for the file as given.
- For the spawn... functions with a *p* suffix, if *pathname* does not contain an explicit directory, the functions will search first the current directory, then the directories set with the DOS PATH environment variable.

strerror *modified*

Name **strerror** – returns pointer to error message string

Usage char *strerror(int *errnum*);

Related functions usage char *_strerror(const char **str*);

Prototype in string.h

Description **strerror** in version 1.5 differs from the same-named function in version 1.0. The new **strerror** has been modified for ANSI compatibility: it now takes an `int` *errnum* parameter, which is an error number, not a string. **strerror** returns a pointer to the error message string associated with error *errnum*.

_strerror allows you to generate customized error messages; it returns a pointer to a null-terminated string containing an error message.

- If *str* is NULL, the return value points to the most-recently generated error message.
- If *str* is not NULL, the return value contains *str* (your customized error message), a colon, a space, the most-recently generated system error message, and a newline. *str* should be 94 characters or less.

_strerror is the same as the old **strerror**, except that *str* is now a `const char *`, instead of a `char *`.

Return value Both functions return a pointer to the string for an error message.

_strerror *misc*

Name **_strerror** – returns pointer to error message string

Usage char *_strerror(const char **string*);

Prototype in string.h

Description see **strerror** in this addendum.

strtoul *misc*

Name **strtoul** – converts a string to an unsigned long

Usage unsigned long strtoul(const char **str*, char ***endptr*, int *radix*);

Prototype in stdlib.h

Description **strtoul** operates the same as **strtol**, except that it converts a string, *str*, to an unsigned long value (whereas **strtol** converts to a long). Refer to the entry for **strtol** (under **str...**) in your *Turbo C Reference Guide* for more information.

Return value **strtoul** returns the converted value, an unsigned long.

tmpnam *misc*

Name **tmpnam** – creates a unique file name

Usage char *tmpnam(char **sptr*);

Prototype in stdio.h

Description **tmpnam** creates a unique file name, which can safely be used as the name of a temporary file. **tmpnam** generates a different string each time you call it, up to TMP_MAX times. TMP_MAX is defined in STDIO.H as 65535.

The parameter to **tmpnam**, *sptr*, is either NULL or a pointer to an array of at least *L_tmpnam* characters: *L_tmpnam* is defined in STDIO.H. If *sptr* is NULL, **tmpnam** leaves the generated temporary file name in an internal static object and returns a pointer to that object.

If *sptr* is not NULL, **tmpnam** places its result in that pointed-to array and returns *sptr*.

Note: If you do create such a temporary file with **tmpnam**, it is your responsibility to delete the file name (for example, with a call to **remove**). It is not deleted automatically.

Return value If *sptr* is NULL, **tmpnam** returns a pointer to an internal static object. Otherwise, **tmpnam** returns *sptr*.

Portability ANSI C, UNIX

See also **creat, fopen, mktemp, open, tmpfile**

tmpfile *misc*

Name **tmpfile** – opens a binary "scratch" file

Usage

```
#include <stdio.h>
FILE *tmpfile(void);
```

Prototype in stdio.h

Description **tmpfile** creates a temporary binary file and opens it for update ("w+b"). The file is automatically removed when it's closed or when your program terminates.

Return value **tmpfile** returns a pointer to the stream of the temporary file created. If the file can't be created, **tmpfile** returns NULL.

Portability ANSI C, UNIX

See also **mktemp, tmpnam**

textattr *text*

Name **textattr** – sets text attributes

Usage void textattr(int *attribute*);

Prototype in conio.h

Description **textattr** lets you set both the foreground and background colors in a single call. (Normally, you set the attributes with **textcolor** and **textbackground.**)

This function does not affect any characters currently on the screen; it only affects those displayed using direct console output functions (such as **cprintf**) *after* this function is called.

The color information is encoded in the *attribute* parameter as follows:

```
                      |
  7    6    5    4 | 3    2    1    0
 __________________|__________________
 |    |    |    |    |    |    |    |    |
 | B | b | b | b | f | f | f | f |
 |___|___|___|___|___|___|___|___|
                      |
```

In this 8-bit *attribute* parameter,

`ffff` is the 4-bit foreground color (0 to 15)
`bbb` is the 3-bit background color (0 to 7)
`B` is the blink-enable bit

If the blink-enable bit is *on*, the character will blink. This can be accomplished by adding the constant BLINK to the attribute.

If you use the symbolic color constants defined in CONIO.H for creating text attributes with **textattr**, note the following limitations on the color you select for the background:

You can only select one of the first eight colors for the background.

You must shift the selected background color left by 4 bits to shift it into the correct bit positions.

(These symbolic constants are listed in a table under the lookup entry for **textbackground**.)

Return value None

Portability This text mode function works on IBM PCs and BIOS-compatible systems, only.

See also **textbackground**, **textcolor**

Example

```
/* select blinking yellow characters on a blue background */

textattr(YELLOW + (BLUE<<4) + BLINK);
cputs("Hello, world");
```

textbackground *text*

Name **textbackground** – selects new text background color

Usage void textbackground(int *color*);

Related functions usage void textcolor(int *color*);

Prototype in conio.h

Description These functions select new colors for the text characters and text background.

textcolor selects the foreground character color.

textbackground selects the background text color.

The foreground (background color) of all characters subsequently written by the console output functions will be the color given by *color*. These functions do not affect any characters currently on the screen, but only affect those displayed using direct console output (such as **cprintf**) after the functions are called.

color is an integer from 0 to 7 for **textbackground**, or from 0 to 15 for **textcolor**. You can give the color using a symbolic constant defined in CONIO.H; if you use these constants, you must `#include <conio.h>`.

The following table lists the allowable colors (as symbolic constants), their numeric values, and whether they are available as foreground and background colors, or just foreground.

Symbolic constant	*Numeric value*	*Foreground or background?*
BLACK	0	Both
BLUE	1	Both
GREEN	2	Both
CYAN	3	Both
RED	4	Both
MAGENTA	5	Both
BROWN	6	Both
LIGHTGRAY	7	Both
DARKGRAY	8	Foreground only
LIGHTBLUE	9	Foreground only
LIGHTGREEN	10	Foreground only
LIGHTCYAN	11	Foreground only
LIGHTRED	12	Foreground only
LIGHTMAGENTA	13	Foreground only
YELLOW	14	Foreground only
WHITE	15	Foreground only
BLINK	128	Foreground only

You can make the characters blink by adding 128 to the foreground color. The pre-defined constant BLINK exists for this purpose. For example,

```
textcolor(CYAN + BLINK);
```

Note: Some monitors do not recognize the intensity signal used to create the eight "light" colors (8-15). On such monitors, the light colors will be displayed as their "dark" equivalents (0-7). Also, systems which do not display in color may treat these numbers as shades of one color, special patterns, or special attributes (such as underlined, bold, italics, etc.). Exactly what you'll see on such systems depends upon your own hardware.

Return value None

Portability These functions work with IBM PCs and compatibles, only; corresponding functions exist in Turbo Pascal.

See also **textattr**

Example

```
textcolor(GREEN);                              /* selects green characters */
textbackground(MAGENTA);                        /* on a magenta background */
```

textcolor *text*

Name **textcolor** – selects new character color in text mode

Usage
```
#include <conio.h>
void textcolor(int color);
```

Prototype in conio.h

Description see **textbackground**

textheight *graphics*

Name **textheight** – returns the height of a string, in pixels

Usage
```
#include <graphics.h>
int far textheight(char far *textstring);
```

Related functions usage
```
int far textwidth(char far *textstring);
```

Prototype in graphics.h

Description

textheight takes the current font size and multiplication factor, and determines the height of *textstring* in pixels.

textwidth takes the string length, current font size, and multiplication factor, and determines the width of *textstring* in pixels.

These functions are useful for for adjusting the spacing between lines, computing viewport heights and widths, sizing a title to make it fit on a graph or in a box, and so on.

For example, with the 8×8 bit-mapped font and a multiplication factor of 1 (set by **settextstyle**), the string `TurboC` is 8 pixels high and 48 pixels wide.

It is important to use **textheight** and **textwidth** to compute the height and width of strings, instead of doing the computations manually. By using these functions, no source code modifications have to be made when different fonts are selected.

Return value **textheight** returns the text height in pixels; **textwidth** returns the text width in pixels.

Portability Similar routines exist in Turbo Pascal 4.0

See also **gettextsettings, outtext**

textmode *text*

Name **textmode** – puts screen in text mode

Usage void textmode(int *mode*);

Prototype in conio.h

Description **textmode** selects a specific text mode.

You can give the text mode (the argument *mode*) by using a symbolic constant from the enumeration type *text_modes* (defined in CONIO.H); if you use these constants, you must `#include <conio.h>`.

The *text_modes* type constants, their numeric values, and the modes they specify are given in the following table.

Symbolic constant	Numeric value	Text mode
LAST	–1	Previous text mode
BW40	0	Black & white, 40 columns
C40	1	Color, 40 columns
BW80	2	Black & white, 80 columns
C80	3	Color, 80 columns
MONO	7	Monochrome, 80 columns

When **textmode** is called, the current window is reset to the entire screen, and the current text attributes are reset to normal, corresponding to a call to **normvideo.**

Specifying LAST to **textmode** causes the most-recently-selected text mode to be reselected. This feature is really only useful when you want to return to text mode after using a graphics mode.

Return value None

Portability This function works with IBM PCs and compatibles, only; a corresponding function exists in Turbo Pascal.

See also **gettextinfo**

textwidth *graphics*

Name textwidth – returns the width of a string, in pixels

Usage #include <graphics.h>
int far textwidth(char far **textstring*);

Prototype in graphics.h

Description see **textheight**

wherex *text*

Name **wherex** – gives horizontal cursor position within window

Usage int wherex(void);

Related funtions usage int wherey(void);

Prototype in conio.h

Description **wherex** returns the *x*-coordinate of the current cursor position (within the current text window). **wherey** returns the *y*-coordinate of the current cursor position (within the current text window).

Return value **wherex** returns an integer in the range 1 to 80.
wherey returns an integer in the range 1 to 25.

Portability These functions work with IBM PCs and compatibles, only; corresponding functions exist in Turbo Pascal.

See also **gotoxy**

Example

```
printf(" The cursor is at (%d,%d)\n", wherex(),wherey());
```

wherey *text*

Name	**wherey** – gives vertical cursor position within window
Usage	int wherey(void);
Prototype in	conio.h
Description	see **wherex**.

window *text*

Name — **window** – defines active text mode window

Usage — void window(int *left*, int *top*, int *right*, int *bottom*);

Prototype in — conio.h

Description — **window** defines a text window on the screen. If the coordinates are in any way invalid, the call to *window()* is ignored.

left and *top* are the screen coordinates of the upper left corner of the window.

right and *bottom* are the screen coordinates of the lower right corner.

The minimum size of the text window is 1 column by 1 line. The default window is full screen, with these coordinates:

80-column mode:	1, 1, 80, 25
40-column mode:	1, 1, 40, 25

Return value — None

Portability — This function works with IBM PCs and compatibles, only; a corresponding function exists in Turbo Pascal.

See also — **gettextinfo, textmode**

write *modified*

Name	**write** – writes to a file
Usage	int write(int *handle*, void **buf*, unsigned *nbyte*);
Related functions usage	int _write(int *handle*, void **buf*, unsigned *nbyte*);
Prototype in	io.h
Description	These functions have the same description as given in the *Turbo C Reference Guide*, with the following additions:
	The maximum number of bytes that either of these functions can write is 65534, since 65535 (0xFFFF) is the same as –1, which is the error return indicator for these functions.

C H A P T E R

5

Revised Function Prototypes

This chapter updates some of the function prototypes listed in Chapter 2 of the *Turbo C Reference Manual*. In some cases, we revised the prototypes to keep up with the Draft Proposed ANSI C Standard. Others needed corrections. Refer to Chapter 4 in this addendum for information about new functions and about old functions whose descriptions have been modified.

Function Name	*Header File*	*Revised Usage*
access	IO.H	int access(const char **filename*, int *amode*);
asctime	TIME.H	char *asctime(const struct tm **tm*);
atof	MATH.H STDLIB.H	double atof(const char **nptr*);
atoi	STDLIB.H	int atoi(const char **nptr*);
atol	STDLIB.H	long atol(const char **nptr*);
brk	ALLOC.H	int brk(void **endds*);
chdir	DIR.H	int chdir(const char **path*);
_chmod	IO.H	int _chmod(const char **filename*, int *func*[, int *attrib*]);
chmod	IO.H	int chmod(const char **filename*, int *permiss*);
_creat	IO.H	int _creat(const char **filename*, int *attrib*);
creat	IO.H	int creat(const char **filename*, int *permiss*);
creatnew	IO.H	int creatnew(const char **filename*, int *attrib*);

Function Name	*Header File*	*Revised Usage*
cscanf	CONIO.H	int cscanf(const char **format*[, *argument*,...]);
ctime	TIME.H	char *ctime(const time_t **clock*);
eof	IO.H	int eof(int *handle*);
farcoreleft	ALLOC.H	unsigned long farcoreleft(void);
findfirst	DIR.H	int findfirst(const char **pathname*, struct ffblk **ffblk*, int *attrib*);
fnmerge	DIR.H	void fnmerge(char **path*, const char **drive*, const char **dir*, const char *name*, const char **ext*);
fnsplit	DIR.H	int fnsplit(const char **path*, char **drive*, char **dir*, char **name*, char **ext*);
fopen	STDIO.H	FILE *fopen(const char **filename*, const char **type*);
_fpreset	FLOAT.H	void _fpreset(void);
fprintf	STDIO.H	int fprintf(FILE **stream*, const char **format*[, *argument*,...]);
fputchar	STDIO.H	int fputchar(int *ch*);
fputs	STDIO.H	int fputs(const char **string*, FILE **stream*);
fread	STDIO.H	size_t fread(void **ptr*, size_t *size*, size_t *nitems*, FILE **stream*)
freopen	STDIO.H	FILE *freopen(const char **filename*, const char **type*, FILE **stream*);
fscanf	STDIO.H	int fscanf(FILE **stream*, const char **format*[, *argument*,...]);
fstat	STAT.H	int fstat(int *handle*, struct stat **buff*);
fwrite	STDIO.H	size_t fwrite(const void **ptr*, size_t *size*, size_t *nitems*, FILE **stream*);
getcwd	DIR.H	char *getcwd(char **buf*, int *n*);
getdfree	DOS.H	void getdfree(unsigned char *drive*, struct dfree **dfreep*);
getenv	STDLIB.H	char *getenv(const char **envvar*);
getfat	DOS.H	void getfat(unsigned char *drive*, struct fatinfo **fatblkp*);

Function Name	*Header File*	*Revised Usage*
getpass	CONIO.H	char *getpass(const char **prompt*);
gmtime	TIME.H	struct tm *gmtime(const time_t **clock*);
inportb	DOS.H	unsigned char inportb(int *port*);
ioctl	IO.H	int ioctl(int *handle*, int *cmd*[, void * *argdx*, int *argcx*]);
keep	DOS.H	void keep(unsigned char *status*, unsigned *size*);
localtime	TIME.H	struct tm *localtime(const time_t **clock*);
memccpy	MEM.H STRING.H	void *memccpy(void **destin*, const void **source*, int *ch*, size_t *n*);
memchr	MEM.H STRING.H	void *memchr(const void **s*, int *ch*, size_t *n*);
memcmp	MEM.H STRING.H	int memcmp(const void **s1*, const void **s2*, size_t *n*);
memcpy	MEM.H STRING.H	void * memcpy(void **destin*, const void **source*, size_t *n*);
memicmp	MEM.H STRING.H	int memicmp(const void **s1*, const void **s2*, size_t *n*);
memmove	MEM.H STRING.H	void * memmove(void **destin*, const void **source*, size_t *n*);
memset	MEM.H STRING.H	void *memset(void **s*, int *ch*, size_t *n*);
mkdir	DIR.H	int mkdir(const char **pathname*);
mktemp	DIR.H	char *mktemp(char **template*);
movedata	MEM.H STRING.H	void movedata(unsigned *segsrc*, unsigned *offsrc*, unsigned *segdest*, unsigned *offdest*, size_t *numbytes*);
_open	IO.H	int _open(const char **pathname*, int *access*);
open	IO.H	int open(const char **pathname*, int *access*[, unsigned *permiss*]);
outportb	DOS.H	void outportb(int *port*, unsigned char *byte*);
parsfnm	DOS.H	char *parsfnm(const char **cmdline*, struct fcb **fcbptr*, int *option*);

Function Name	*Header File*	*Revised Usage*
peek	DOS.H	int peek(unsigned *segment*, unsigned *offset*);
peekb	DOS.H	char peekb(unsigned *segment*, unsigned *offset*);
perror	STDIO.H	void perror(const char **s*);
poke	DOS.H	void poke(unsigned *segment*, unsigned *offset*, int *value*);
pokeb	DOS.H	void pokeb(unsigned *segment*, unsigned *offset*, char *value*);
printf	STDIO.H	int printf(const char **format*[, *argument*, ...]);
putenv	STDLIB.H	int putenv(const char **envvar*);
puts	STDIO.H	int puts(const char **string*);
qsort	STDLIB.H	void qsort(void **base*, size_t *nelem*, size_t *width*, int (**fcmp*) (const void *, const void *));
rename	STDIO.H	int rename(const char **oldname*, const char **newname*);
rewind	STDIO.H	void rewind(FILE **stream*);
rmdir	DIR.H	int rmdir(const char **pathname*);
sbrk	ALLOC.H	void *sbrk(int *incr*);
scanf	STDIO.H	int scanf(const char **format*[, *argument*, ...]);
searchpath	DIR.M	char *searchpath(const char **filename*);
setblock	DOS.M	int setblock(unsigned *seg*, unsigned *newsize*);
setmode	IO.H	int setmode(int *handle*, int *mode*);
setvbuf	STDIO.H	int setvbuf(FILE **stream*, char **buf*, int *type*, size_t *size*);
sleep	DOS.H	void sleep(unsigned *seconds*);
sprintf	STDIO.H	int sprintf(char **string*, const char **format* [, *argument*, ...]);
sscanf	STDIO.H	int sscanf(char **string*, const char **format* [, *argument*, ...]);
_status87	FLOAT.H	unsigned int _status87(void);
stime	TIME.H	int stime(time_t **tp*);

Function Name	*Header File*	*Revised Usage*
stpcpy	STRING.H	char *stpcpy(char **destin*, const char **source*);
strcat	STRING.H	char *strcat(char **destin*, const char **source*);
strchr	STRING.H	char *strchr(const char **str*, int *c*);
strcmp	STRING.H	int strcmp(const char **str1*, const char **str2*);
strcpy	STRING.H	char *strcpy (char **destin*, const char **source*);
strcspn	STRING.H	size_t strcspn(const char **str1*, const char **str2*);
strdup	STRING.H	char *strdup(const char **str*);
stricmp	STRING.H	int stricmp(const char **str1*, const char **str2*);
strlen	STRING.H	size_t strlen(const char **str*);
strncat	STRING.H	char *strncat(char **destin*, const char **source*, size_t *maxlen*);
strncmp	STRING.H	int strncmp(const char **str1*, const char **str2*, size_t *maxlen*);
strncpy	STRING.H	char *strncpy(char **destin*, const char **source*, size_t *maxlen*);
strnicmp	STRING.H	int strnicmp(const char **str1*, const char **str2*, size_t *maxlen*);
strnset	STRING.H	char *strnset(int **str*, int *ch*, size_t *n*);
strpbrk	STRING.H	char *strpbrk(const char **str1*, const char **str2*);
strrchr	STRING.H	char *strrchr(const char **str*, int *c*);
strset	STRING.H	char *strset(char **str*, int *ch*);
strspn	STRING.H	size_t strspn(const char **str1*, const char **str2*);
strstr	STRING.H	char *strstr(const char **str1*, const char **str2*);
strtod	STDLIB.H	double strtod(const char **str*, char ***endptr*);
strtok	STRING.H	char *strtok(char **str1*, const char **str2*);
strtol	STDLIB.H	long strtol(const char **str*, char ***endptr*, int *base*);
system	PROCESS.H STDLIB.H	int system(const char **command*);

Function Name	*Header File*	*Revised Usage*
time	TIME.H	time_t time(time_t **tloc*);
ungetc	STDIO.H	int ungetc(int *c*, FILE **stream*);
unlink	DOS.H IO.H	int unlink(const char **filename*);
vfprintf	STDIO.H	int vfprintf(FILE **stream*, const char **format*, va_list *param*);
vfscanf	STDIO.H	int vfscanf(FILE **stream*, const char **format*, va_list *param*);
vprintf	STDIO.H	int vprintf(const char **format*, va_list *param*);
vscanf	STDIO.H	int vscanf(const char **format*, va_list *param*);
vsprintf	STDIO.H	int vsprintf(char **string*, const char **format*, va_list *param*);
vsscanf	STDIO.H	int vsscanf(char **string*, const char **format*, va_list *param*);

C H A P T E R

6

Miscellaneous Information

In this chapter we document miscellaneous changes to the software, along with changes and additions to the *Turbo C User's Guide* and the *Turbo C Reference Guide*. This information doesn't fall under any of the categories covered in the other chapters and appendixes in this addendum. The information presented in this chapter covers:

- TCCONFIG.EXE (formerly called CNVTCFG.EXE)
- BUILTINS.MAK
- streams
- configuration files
- pick lists and pick files
- corrections to the original manuals

The TCCONFIG.EXE Conversion Utility for Configuration Files

The integrated environment and command-line compiler have a number of common options, listed in Table 2.2 of Appendix C in the *Turbo C Reference Guide*. TCCONFIG.EXE takes a configuration file created by one environment and converts it for use by the other.

The conversion command is

```
TCCONFIG SourceFile [DestinationFile]
```

TCCONFIG automatically determines the direction of the conversion: It examines the source file to see whether it is an integrated environment (TC) configuration file or a command-line compiler (TCC) configuration file.

The destination file name is optional. If you don't specify a file name, TCCONFIG uses the default name TURBOC.TC or TURBOC.CFG, depending on the conversion direction. You can give any file name; however, the command-line compiler only looks for a file named TURBOC.CFG when running. It won't run on any other name.

The TURBOC.TC file uses default values for any items not specified by the command-line compiler configuration file (TURBOC.CFG). In addition, only the options in TURBOC.TC that differ from the default values are included in TURBOC.CFG.

TCCONFIG returns you to the DOS prompt when the conversion is done.

How MAKE Searches for BUILTINS.MAK

BUILTINS.MAK is an optional file in which you can store MAKE macros and rules that you use again and again, so you don't have to keep typing them into your makefiles.

The first place MAKE searches for BUILTINS.MAK is the current directory. If it's not there, *and* if you're running under DOS 3.*x*, MAKE will then search the start directory (where MAKE.EXE resides).

What Are Streams?

Streams are the most portable means for reading or writing data using Turbo C. They are designed to allow flexible and efficient input and output that are not affected by the underlying file or device hardware.

A stream is a file or physical device that you manipulate with a pointer to a FILE object (defined in STDIO.H). The FILE object contains various information about the stream, including the current position of the stream, pointers to any associated buffers, and error or end-of-file indicators.

Your program should never create or copy FILE objects themselves; instead, it should use the pointers returned from functions like **fopen**. Be sure that you do not confuse FILE pointers with DOS file handles (which are used in low-level DOS or UNIX-compatible I/O).

You must first "open" a stream before you can perform I/O on the stream. Opening the stream connects it to the named DOS file or device. The routines that open streams are **fopen**, **fdopen**, and **freopen**. When you open a stream, you indicate whether you want to read or write to the stream, or do both. You also indicate whether you will treat the data of that stream as text or binary data. This last distinction is important because of a minor incompatibility between C stream I/O and DOS text files.

Text vs. Binary Streams

Text streams are used for normal DOS text files, such as a file created with the Turbo C editor. C stream I/O assumes that text files are divided into lines separated by a single newline character (which is the ASCII line-feed). DOS text files, however, are stored on disk with two characters between each line, an ASCII carriage-return and a line-feed. In text mode, Turbo C translates carriage-return line-feed (CR/LF) pairs into a single line-feed on input; line-feeds are translated to CR/LF pairs on output.

Binary streams are much simpler than text streams. No such translations are performed. Any character is read or written without change.

A file can be accessed in either text or binary mode without any problems as long as you are aware of and understand the translations taking place in text streams. Turbo C doesn't "remember" how a file was created or last accessed.

If no translation mode is specified when a stream is opened, it is opened in the default translation mode given by the global variable *_fmode*. By default, *_fmode* is set to text mode.

Buffering Streams

Streams are typically buffered when associated with files. This allows I/O at the individual character level—such as with **getc** and **putc**—to be very fast. You can supply your own buffer, change the size of the buffer used, or force the stream to use no buffer at all by calling **setvbuf** or **setbuf**.

Buffers are automatically flushed when the buffer is full, the stream is closed, or the program terminates normally. You can use **fflush** and **flushall** to force the buffers to be flushed manually.

Normally, you use streams to sequentially read or write data. I/O takes place at the current file position. Whenever you read or write data, the program moves the file position to immediately after the just-accessed data.

A stream that is connected to a disk file can also be randomly accessed. You can use **fseek** to position a file, then issue several read or write operations to access the data after that point.

When you are both reading and writing data to a stream, you should not freely mix reading and writing operations. You must flush the stream's buffer between reading and writing data. A call to **fflush, flushall,** or **fseek** clears the buffer and allows you to switch operations. For maximum portability, you should flush even when no buffer is present, since other systems may have additional restrictions on mixing input and output operations even without a buffer.

Predefined Streams

In addition to streams created by calling **fopen**, five predefined streams are available whenever your program begins execution. The following names correspond to these streams:

Name	*I/O*	*Mode*	*Stream*
stdin	Input	Text	Standard Input
stdout	Output	Text	Standard Output
stderr	Output	Text	Standard Error
stdaux	Both	Binary	Auxiliary I/O
stdprn	Output	Binary	Printer Output

The *stdaux* and *stdprn* streams are specific to DOS and are not portable to other systems.

The *stdin* and *stdout* streams can be redirected by DOS, while the others are connected to specific devices: *stderr* to the console (CON:), *stdprn* to the printer (PRN:), and *stdaux* to the auxiliary port.

The auxiliary port depends on your machine's configuration; it is typically COM1:. Consult your DOS documentation for information about redirecting input or output on a DOS command line. If not redirected, *stdin* and *stdout* are connected to the console (CON: device). Furthermore, if not redirected, *stdin* is line buffered, while *stdout* is unbuffered. The other predefined streams are unbuffered.

To process a predefined stream in a mode other than its default (for example, to process *stdprn* in text mode), use **setmode**. The predefined

stream names are constants; you cannot assign values to them. If you want to reassociate one of them to a file or device, use **freopen**.

What is a Configuration File?

Basically, a configuration file is a file that contains information pertinent to Turbo C. In it, you store such information as your selected compiler options, your linker options, various directories that Turbo C will need to search when compiling and linking your programs, and so on.

There are two types of Turbo C configuration files: one you use with TCC.EXE (command-line Turbo C), and the other you use with TC.EXE (the Turbo C integrated environment). There is only one command-line configuration file; it's named TURBOC.CFG. The integrated environment configuration file can have any file name. The file TCCONFIG.TC is the default (assumed) integrated environment configuration file.

In this section we briefly summarize the command-line configuration file (TURBOC.CFG), then refer you to other documentation in the *Turbo C User's Guide*. After that, we cover the integrated environment configuration files in detail.

The TURBOC.CFG Configuration File

When you invoke command-line Turbo C, it looks for a file named TURBOC.CFG. Such a command-line configuration file, if it exists, can contain any of the Turbo C compiler command-line options.

If you've listed your commonly-used options in TURBOC.CFG, you won't need to enter them on the command line when you use TCC.EXE. If you don't want to use certain options that are listed in TURBOC.CFG, you can override them with switches on the command line.

For more information about TURBOC.CFG, refer to "The TURBOC.CFG File" in Chapter 3 of the *Turbo C User's Guide.*

The TC Configuration Files

When you start using the Turbo C integrated environment for the first time, there is no configuration file. TC.EXE will start up with all the menu items set to their internal defaults (Memory model will be set to Small, Calling

convention set to C, Keep messages set to No, etc.). In the course of using the integrated environment, you will probably want to change some of the menu items' settings.

If you exit Turbo C without saving the new settings in a configuration file, then the next time you invoke the integrated environment, it will again start up with all the menu items set to their previous defaults. But if, instead, you save the new settings to a configuration file, then the next time the integrated environment starts up, the menu items will be set to the values you chose, and you won't have to go through the process of resetting them.

TCCONFIG.TC

When you start up TC.EXE, it looks for a configuration file named TCCONFIG.TC. It looks for that file in certain locations (we'll explain exactly where it looks later); if TC.EXE can't find a TCCONFIG.TC file, the integrated environment starts up using the default settings that are built into TC.EXE.

Other TC Configuration Files

You can also start up TC.EXE at the DOS prompt with a request for a specific configuration file, using the `/c` option. For example, if you type

```
tc /cmyconfig
```

at the DOS prompt, Turbo C will look for a configuration file named MYCONFIG.TC in the current directory (if you give no extension, Turbo C assumes the extension .TC).

If Turbo C can't find the configuration file you named, it will issue a warning message to that effect. It won't look for any other configuration file, but it will still start up, using the built-in default settings.

What is Stored in TC Configuration Files?

The information stored in the TC configuration files can be broken down into two categories: compiler-linker options and TC.EXE-specific values.

The compiler-linker options govern the compiler and linker, and they all have corresponding options in the command-line version of Turbo C, while the TC.EXE-specific values are related to the integrated environment itself. Some examples of these values specific to the integrated environment are Project name, Pick file name, and the Environment options.

Creating a TC Configuration File

So how do you create a TC configuration file? Unlike the command-line configuration file (TURBOC.CFG), the integrated environment configuration file is not one you can create or modify with an editor. You must select the Store options item from the Options menu, and then the integrated environment will create the configuration file for you.

Changing Configuration Files Mid-stream

It's easy to change to a different .TC configuration file from within the integrated environment. To do this:

- Select Restore options from the Options menu. A pop-up box will appear, displaying the last configuration file name you typed (it defaults to `*.tc` the first time).
- You can type in a mask (like `*.tc` or `??config.*`) then press *Enter* to bring up a directory listing of .TC files. You then select a file from the directory list.
- Or you can type in a specific configuration file name (then press *Enter* to load that file).

Where Does TC.EXE Look for TCCONFIG.TC?

There are two places TC.EXE will look for the default configuration file TCCONFIG.TC. First, it will search the default (current working) directory. If it does not find TCCONFIG.TC there, it will then search the Turbo C directory, ***if you have previously set the Turbo C directory using TCINST.***

To find out more about the Turbo C directory and TCINST, read Appendix A, "The New TCINST," in this addendum.

TCINST vs. the Configuration File: Who's the Boss?

You can use TCINST to set any the items found on Turbo C's Options/Directories or Options/Environment menu, and then store those settings directly in TC.EXE. If there is no TC configuration file to be found when you start up that customized TC.EXE, those settings you customized will be the defaults.

However, if TC.EXE starts up and finds a TCCONFIG.TC file in the default directory (or in the Turbo C directory), that configuration file's settings will take precedence over any default settings you installed with TCINST.

Also, if you invoke TC.EXE with a `/c` option, and Turbo C finds the configuration file you specified, that file's settings will take precedence over the TCINST-installed defaults.

What Does "Config auto save" Do?

Normally, Turbo C will save the current configuration file (write it out to disk) only when you give the Options/Store options command. However, you can direct Turbo C to automatically save the configuration file under additional circumstances.

Just toggle the Options/Environment/Config auto save menu item to *on*. With Config auto save on, Turbo C will also save the file whenever you select Run or File/OS shell, or when you exit the integrated environment (by selecting File/Quit)—if the configuration file has never been saved, or if it has been at all modified since it was last saved.

With Config auto save on, if the configuration file has not yet been saved, Turbo C will choose a file name for the auto saved file. The chosen name is the last configuration file you stored or retrieved, or TCCONFIG.TC (in the current directory) if you haven't loaded, retrieved or saved a configuration file yet.

What are Pick Lists and Pick Files?

The *pick list* and *pick file* are two features of the Turbo C integrated environment that work together to save the state of your editing sessions. The pick list remembers what files you are editing *while you are in* the integrated environment. The pick file remembers what files you were editing *after you leave* the integrated environment or *after you change contexts within* the integrated environment. (Changing contexts encompasses loading a new configuration file or defining a new pick file name.)

The Pick List

The pick list is a menu located in the File menu; you call it up by selecting File/Pick or by pressing the *Alt-F3* hot key. The pick list provides a list of the

eight files most recently loaded into the editor. The top file listed is the file currently in the editor. If there is more than one file name in the pick list, the second file name listed is highlighted; this is the file just previously loaded into the editor.

To load a file from the pick list into the editor, scroll the selection bar to highlight the appropriate file name, then press *Enter*. When you do this, Turbo C will load the selected file into the editor, then the editor will position the cursor in that newly-loaded file at the location you last left it. In addition, any marked block and markers in that file will be exactly as you left them.

The pick list is a handy tool for moving back and forth between your files as you develop your program. By pressing *Alt-F3 Enter* in succession, you can alternate between two files.

If the file you want is not on the pick list, you can select `--load file--` (the last entry on the pick list menu). This will bring up a `Load File Name` input box, and you can type in the name of the file you want (using DOS-style wildcards if necessary). You can also press the *F3* hot key to automatically select File/Load.

The Pick File

The pick file stores editor-related information, including the contents of the pick list. For each entry (file) in the pick list, Turbo C stores the file name, cursor position, marked block, and markers.

In addition to information about each file, the pick file contains data on the state of the editor when you last exited. This includes the most recent search-and-replace strings and search options.

To create a pick file, you must define a pick file name. You can do this by entering a file name in the Pick file name menu item on the Options/Directories menu. If you have defined a pick file name, then whenever you exit the integrated environment, Turbo C updates that pick file on disk.

When and How Do You Get a Pick File?

There are two items on the Options/Directories menu that you can look to for information about the pick file: Pick file name and Current pick file.

Q: How do you know if you already have a pick file?

A: You have a pick file if the Current pick file menu item is not blank.

Q: How did that file name appear in Current pick file?
A: In one of two ways: Either a file name is explicitly listed in Pick file name, or (if Pick file name is blank) you loaded a default pick file.

Q: Suppose the Pick file name item explicitly lists a file name. How did that file name get there?
A: You get a file name in Pick file name by:

1. entering it yourself in the current session, or

2. entering it in a previous session, saving the configuration file, then using that configuration file in the current session, or

3. installing it with TCINST

Q: Suppose Pick file name is blank, but Current pick file is not blank. How did that default pick file get loaded?
A: There was a default pick file, TCPICK.TCP, in the current directory or (if not there) in the Turbo C directory, and Turbo C loaded it automatically on start-up.

Once a pick file is loaded, the integrated environment remembers the full path name. This information is displayed in the Current pick file menu item.

When Does Turbo C Save Pick Files?

Turbo C saves the file named in Current pick file whenever you exit the integrated environment. In addition, any time the pick file name is changed (either directly by entering a new name from the menu item, or indirectly by loading a configuration file that contains a different pick file name) Turbo C first saves the existing pick file.

Turbo C will *not* save a pick file to disk when you exit if the Current pick file menu item is blank.

Corrections to the Original Manuals

The following list shows minor corrections to the *Turbo C User's Guide* and *Turbo C Reference Guide*. Before going on, check the row of numbers at the bottom of the copyright page in those manuals (the page behind the title page at the front of the manuals). If the numbers are 10 9 8 7 6 5 4 3 2 1,

refer to the page numbers listed here in the first column; if the numbers are 10 9 8 7 6 (and possibly 5), refer to the page numbers listed here in the second column.

Turbo C User's Guide

9	9	Delete the phrase "As we explained in the "Introduction,"."
30	34	Change "Appendix A describes the editor commands...." to "Appendix A of the *Turbo C Reference Guide* decribes the editor commands...."
62	70	In the example code for MYMAIN.C, insert `char *GetString(void);` between the `#include` statement and the beginning of `main()`
164	175	The line of code that says `strcpy(current.last = "Smith");` should say `strcpy(current.last,"Smith");`
165	175	Replace `pstudent -> last = "Jones";` with `strcpy(pstudent -> last,"Jones");`
174	183	Delete the curly bracket (`}`) after `fclose(f);`
187	195	In the `zwf()` statement, change `parm1=%d` and `parm2=%d` to `parm1=%f` and `parm2=%f`, respectively.
235	244	In the description of **Compact**, the phrase "The inverse of medium" does not imply that 1 Mb of static data is possible.

Turbo C Reference Guide

——	19	In line five, the phrase "the with double quotes" should read "them with double quotes".
18	24	*_fmode* Usage: Replace "int" with "unsigned".
41	50	**bioskey** Description: Replace "BIOS interrupt 0x14" with "Bios interrupt 0x16".
60	70	**_creat** Description: First line, change "_create" to "_creat" (no ending **e**).
66	76	**dosexterr** Usage and Description: "DOSERR" should be "DOSERROR".

68	78	**dup** Description: Change "**dup2** returns the next file handle available" to "**dup** returns the next file handle available".
70	80	**eof** Usage: Delete " * " in " **handle*".
118	127	**getftime** Usage and Prototype in: Replace "<dos.h>" with "<io.h>".
190	200	**rename** Return value: Replace "ENOTSAME Not same device" with "EXDEV Cross-link device".
——	209	**...scanf** %[search_set] conversion: Last two examples on page; replace "A through Z" with "A through F" (twice).
——	210	**...scanf** %[search_set] conversion: Example at top of page: replace "A through Z" with "A through F".
256	268	Description of **Delete character under cursor**: Change "This command does not work..." to "This command works...".
318	331	Under ***"Full File name Macro ($<),"*** change both occurrences of `.obj.c:` to `.c.obj:`

A P P E N D I X

A

The New TCINST

In this appendix, we cover the new version of the Turbo C Installation (or customization) program, TCINST.EXE, which is included in your Turbo C 1.5 package.

The first thing you should do after copying the Turbo C 1.5 files to your system is either delete or rename the old TCINST.COM. This is necessary because TCINST for version 1.5 is now an .EXE file. If both the .COM and .EXE TCINST files are on disk and you type `tcinst` *Enter*, DOS will find TCINST.COM (from version 1.0) instead of TCINST.EXE (from version 1.5).

Appendix F in the *Turbo C Reference Guide* covers the original TCINST.COM, shipped with Turbo C version 1.0. The original TCINST.COM *cannot* be used with Turbo C version 1.5, and the new TCINST.EXE *cannot* be used with Turbo C version 1.0. Don't worry, though: TCINST will reject a mismatched version of TC.EXE with an error message. For information about the new TCINST, read *this* appendix, not Appendix F.

What Is TCINST?

TCINST is the Turbo C Installation program; you use it to customize TC.EXE, the integrated development environment version of Turbo C. Through TCINST, you can change various default settings in the TC operating environment, such as the screen size, editing modes, menu colors, and default directories. TCINST lets you change the environment in which you operate Turbo C: It directly modifies certain default values within your copy of TC.EXE.

With TCINST, you can do any of the following:

- set up paths to the directories where your include, library, configuration, Help, pick, and output files are located
- customize the editor command keys
- set up Turbo C's editor defaults and on-screen appearance
- set up the default video display mode
- change screen colors
- resize Turbo C's Edit and Message windows

Turbo C comes ready to run: There is no installation *per se*. You can copy the files from the distribution disks to your working floppies (or hard disk), as described in Chapter 1 of the *Turbo C User's Guide*, then run Turbo C. However, you will need to run TCINST if you want to change the defaults directly in TC.EXE.

If you want to store path names (to all the different directories you use when running TC) directly in TC.EXE, you'll need to use the Turbo C directories option.

You can use the Editor commands option to reconfigure (customize) the interactive editor's keystrokes to your liking.

The Setup environment option is for setting various values that have to do with the default editing modes and the appearance of the TC integrated environment.

With Display mode, you can specify the video display mode that TC will operate in, and whether yours is a "snowy" video adapter.

You can customize the colors of almost every part of TC's integrated environment through the Colors option.

The Resize windows option allows you to change the sizes of the Edit and Message windows.

Running TCINST

1) The syntax for TCINST is

```
tcinst [/c] [pathname]
```

Both *pathname* and `/c` are optional. If *pathname* is not supplied, TCINST looks for TC.EXE in the current directory. Otherwise, it uses the given path name. Normally, TCINST comes up in black and white, even on a color monitor. If you want to run TCINST in color, give the `/c` option.

Note: You can use one version of TCINST to customize several different copies of Turbo C on your system. These various copies of TC.EXE can have different executable program names; all you need to do is invoke TCINST and give a path name to the copy of TC.EXE you're customizing; for example,

```
tcinst tc.exe

tcinst ..\..\bwtc.exe

tcinst /c c:\borland\colortc.exe
```

In this way, you can customize the different copies of Turbo C on your system to use different editor command keys, different menu colors, and so on.

2) From the main TCINST installation menu, you can select Turbo C directories, Editor commands, Setup environment, Display mode, Colors, Resize windows, or Quit/save.

   You can either press the highlighted capital letter of a given option, or use the *Up* and *Down* arrow keys to move to your selection and then press *Enter*. For instance, press *C* to modify the Colors of the TC integrated environment.

3) In general, pressing *Esc* (more than once if necessary) returns you from a submenu to the main installation menu.

The Turbo C Directories Option

With Turbo C directories, you can specify a path to each of the TC.EXE default directories. These are the directories Turbo C searches when looking for an alternate configuration file, the Help file, the include and library files, and the directory where it will place your program output.

When you select Turbo C directories, TCINST brings up a submenu. The items on this submenu are

- Include directories
- Library directories
- Output directory
- Turbo C directory
- Pick file name

You enter names for each of these just as you do for the corresponding menu items in TC.EXE. If you are not certain of each item's syntax, refer first to Chapter 2 in this addendum and then Chapter 2 in the *Turbo C User's Guide*.

Include directories and Library directories:

You can enter multiple directories in Include directories and Library directories: You must separate the directory path names with a semicolon (;), and you can enter a maximum of 127 characters with either menu item. You can enter absolute or relative path names.

Output directory and Turbo C directory:

The Output directory and Turbo C directory menu items each take one directory path name; each item accepts a maximum of 64 characters.

The Turbo C directory is where TC looks for the Help file and TCCONFIG.TC (the default configuration file) if they aren't in the current directory.

Pick file name:

When you select this menu item, an input window pops up. In it, you type the path name of the Pick file you want Turbo C to load or create. There is no default installed Pick file name.

After typing a path name (or names) for any of the Setup environment menu items, press *Enter* to accept, then press *Esc* to return to the main TCINST installation menu. When you exit the program, TCINST prompts you on whether you want to save the changes. Once you save the Turbo C directories paths, the locations are written to disk and become part of TC.EXE's default settings.

The Editor Commands Option

Turbo C's interactive editor provides many editing functions, including commands for

- cursor movement
- text insertion and deletion
- block and file manipulation
- string search (plus search-and-replace)

These editing commands are assigned to certain keys (or key combinations): They are explained in detail in Appendix A of the *Turbo C Reference Guide.*

When you select Editor commands from TCINST's main installation menu, the `Install Editor` screen comes up, displaying three columns of text.

- The first column (on the left) describes all the functions available in TC's interactive editor.
- The second column lists the *Primary* keystrokes: what keys or special key combinations you press to invoke a particular editor command.
- The third column lists the *Secondary* keystrokes: These are optional alternate keystrokes you can also press to invoke the same editor command.

Note: Secondary keystrokes always take precedence over primary keystrokes.

The bottom lines of text in the `Install Editor` screen summarize the keys you use to select entries in the Primary and Secondary columns.

Key	*Legend*	*What It Does*
Left, Right Up and *Down* arrow keys	select	Selects the editor command you want to re-key.
Page Up and *Page Down* arrow keys	page	Scrolls up or down one full screen page
Enter	modify	Enters the keystroke-modifying mode.
R	restore factory defaults	Resets all editor commands to the factory default keystrokes.
Esc	exit	Leaves the `Install Editor` screen and returns to the main TCINST installation menu.
F4	Key Modes	Toggles between the three flavors of keystroke combinations.

After you press *Enter* to enter the `modify` mode, a pop-up window lists the current defined keystrokes for the selected command, and the bottom lines of text in the `Install Editor` screen summarize the keys you use to change those keystrokes.

Key	*Legend*	*What It Does*
Backspace	backspace	Deletes keystroke to left of cursor
Enter	accept	Accepts newly defined keystrokes for selected editor command.
Esc	abandon changes	Abandons changes to the current selection, restoring the command's original keystrokes, and returns to the Install Editor screen (ready to select another editor command).
F2	restore	Abandons changes to current selection, restoring the command's original keystrokes, but keeps the current command selected for re-definition.
F3	clear	Clears current selection's keystroke definition, but keeps the current command selected for re-definition.
F4	Key Modes	Toggles between the three flavors of keystroke combinations: **`WordStar-like`**, **`Ignore case`**, and **`Verbatim`**.

Note: To enter the keys *F2* , *F3* , or *F4* as part of an editor command key sequence, first press the backquote (') key, then the appropriate function key.

Keystroke combinations come in three flavors: **`WordStar-like`**, **`Ignore case`**, and **`Verbatim`**. These are listed on the bottom line of the screen; the highlighted one is the flavor of the current selection. In all cases, the first character of the combination must be a special key or a control character. The combination flavor governs how the subsequent characters are handled.

- `WordStar-like`: In this mode, if you type a letter or the character *[*, *]*, \, *^*, or *–*), it is automatically entered as a Control-Character combination. For example,

typing *a* or *A* or *Ctrl A*	yields	*< Ctrl A >*
typing *y* or *Y* or *Ctrl y*	yields	*< Ctrl Y >*
typing *[*	yields	*< Ctrl [>*

  For example, if you customize an editor command to be *< Ctrl A > < Ctrl B >* in WordStar-like mode, you can type any of the following in the TC editor to activate that command:

  < Ctrl A > < Ctrl B >
  < Ctrl A > B
  < Ctrl A > b

- `Ignore case`: In this mode, all alpha (letter) keys you enter are converted to their uppercase equivalents. When you type a letter in this mode, it is *not* automatically entered as a Control-Character combination; if a keystroke is to be a Control-Letter combination, you must hold down the *Ctrl* key while typing the letter. For example, in this mode, *< Ctrl A > B* and *< Ctrl A > b* are the same, but differ from *< Ctrl A > < Ctrl B >*.
- `Verbatim`: If you type a letter in this mode, it is entered exactly as you type it. So, for example, *< Ctrl A > < Ctrl B >*, *< Ctrl A > B*, and *< Ctrl A > b* are all distinct.

Allowed Keystrokes

Although TCINST provides you with almost boundless flexibility in customizing the Turbo C editor commands to your own tastes, there are a few rules governing the keystroke sequences you can define. Some of the rules apply to any keystroke definition, while others come into effect only in certain keystroke modes.

1. You can enter a maximum of six keystrokes for any given editor command. Certain key combinations are equivalent to two keystrokes: These include *Alt (any valid key)*; the cursor-movement keys (*Up, Page Down, Del,* etc.); and all function keys or function key combinations (*F4, Shift-F7, Alt-F8,* etc.).
2. The first keystroke must be a character that is non-alphanumeric and non-punctuation: i.e., it must be a Control key or a special key.
3. To enter the *Esc* key as a command keystroke, type *Ctrl [*
4. To enter the *Backspace* key as a command keystroke, type *Ctrl H*

5. To enter the *Enter* key as a command keystroke, type *Ctrl M*
6. The Turbo C predefined Help function keys (*F1* and *Alt F1*) can't be reassigned as Turbo C editor command keys. Any other function key can, however. If you enter a Turbo C hot key as part of an editor command key sequence, TCINST will issue a warning that you are overriding a hot key in the editor and verify that you want to override that key. Chapter 2 of the *Turbo C User's Guide* contains a complete list of Turbo C's predefined hot keys.

The Setup Environment Option

You can install several editor default modes of operation with this option: eight of the items on this menu are toggles, and the ninth one brings up a submenu. If you are not familiar with these items, refer first to Chapter 2 in this addendum and then Chapter 2 in the *Turbo C User's Guide.*

The items on the Setup environment menu and their significance are described here.

Backup source files

(*on* by default) With Backup source files *on*, Turbo C automatically creates a backup of your source file when you do a File/Save. It uses the same file name, and adds a .BAK extension: the backup file for *filename*.C would be *filename*.BAK. With Backup source files *off*, no .BAK file is created.

Edit auto save

(*on* by default) With Edit auto save *on*, Turbo C automatically saves the file in the editor (if it's been modified since last saved) whenever you use Run or File/OS shell. This helps prevent loss of your source files in the event of some calamity. With Edit auto save *off*, no such automatic saving occurs.

Config auto save

(*on* by default) With Config auto save *on*, Turbo C automatically saves the configuration file (if it's been modified since last saved) whenever you use Run, File/OS shell, or File/Quit.

Zoom state

With Zoom state *on*, Turbo C starts up with the Edit window occupying the full screen; when you switch to the Message window, it will also be full-screen. With Zoom state *off*, the Edit window occupies the top portion of the screen, above the Message window. (You can resize the

windows with the Resize windows option from the main TCINST installation menu.)

Insert mode

(*on* by default.) With Insert mode *on*, the editor inserts anything you enter from the keyboard at the cursor position, and pushes existing text to the right of the cursor even further right. Toggling Insert mode *off* allows you to overwrite text at the cursor.

Autoindent mode

(*on* by default.) With Autoindent mode *on*, the cursor returns to the starting column of the previous line when you press *Enter*. When Autoindent mode is toggled *off*, the cursor always returns to column one.

Use tabs

(*on* by default.) With Use tabs *on*, when you press the *Tab* key, the editor places a tab character (`^I`) in the text using the tab size specified with `Tab size`. With Use tabs *off*, when you press the *Tab* key, the editor inserts enough space characters to align the cursor with the first letter of each word in the previous line.

Screen size

When you select Screen size, a three-item submenu pops up. With the items in this menu, you can set the Turbo C integrated environment display to one of three sizes (25-, 43-, or 50-line). The available sizes depend on your hardware: 25-line mode is always available; 43-line mode is for systems with an EGA, while 50-line mode is for VGA-equipped systems.

Look at the Quick-Ref line for directions on how to select these options. You can change the operating environment defaults to suit your preferences (and your monitor) then save them as part of Turbo C. Of course, you'll still be able to change these settings from inside Turbo C's editor (or from the Options/Environment menu).

Note: Any option that you install with TCINST that *also* appears as a menu-settable option in TC.EXE will be overridden whenever you load a configuration file that contains a different setting for that option.

The Display Mode Option

Normally, Turbo C correctly detects your system's video mode. You should only change the Display mode option if one of the following holds true:

- You want to select a mode other than the current video mode.
- You have a Color Graphics Adapter that doesn't "snow".
- You think Turbo C is incorrectly detecting your hardware.
- You have a laptop or a system with a composite screen (which acts like a CGA with only one color). For this situation, select Black and white.

Press *D* to select Display mode from the installation menu. A pop-up menu appears; from this menu, you can select the screen mode Turbo C will use during operation. Your options include Default, Color, Black and white, or Monochrome. These are fairly intuitive.

Default

By default, Turbo C always operates in the mode that is active when you load it.

Color

Turbo C uses 80-column color mode, no matter what mode is active when you load TC.EXE, and switches back to the previously active mode when you exit.

Black and white

Turbo C uses 80-column black and white mode characters, no matter what mode is active, and switches back to the previously active mode when you exit. Use this with laptops and composite monitors.

Monochrome

Turbo C uses monochrome mode, no matter what mode is active, and switches back to the previously active mode when you exit.

When you select one of the first three options, the program conducts a video test on your screen; refer to the Quick-Ref line for instructions on what to do. When you press any key, a window comes up with the query

```
Was there Snow on the screen?
```

You can choose

- Yes, the screen was "snowy"
- No, always turn off snow checking
- Maybe, always check the hardware

Look at the Quick-Ref line for more about Maybe. Press *Esc* to return to the main installation menu.

The Color Customization Option

Pressing *C* from the main installation menu allows you to make extensive changes to the Colors of your version of Turbo C. After you press *C*, a menu with these options appears:

- Customize colors
- Default color set
- Turquoise color set
- Magenta color set

Because there are nearly 50 different screen items that you can color-customize, you will probably find it easier to choose a *preset* set of colors to your liking.

There are three preset color sets to choose from. Press *D*, *T*, or *M*, and scroll through the colors for the Turbo C screen items using the *PgUp* and *PgDn* keys. If none of the preset color sets is to your liking, you can design your own.

To make custom colors, press *C* to Customize colors. Now you have a choice of 12 types of items that can be color-customized in Turbo C; some of these are text items, some are screen lines and boxes. Choose one of these items by pressing a letter *A* through *L*.

Once you choose a screen item to color-customize, you will see a pop-up menu and a *view port*. The view port is an example of the screen item you chose, while the pop-up menu displays the components of that selection. The view port also reflects the change in colors as you scroll through the color palette.

For example, if you choose *H* to customize the colors of Turbo C's error boxes, you'll see a new pop-up menu with the four different parts of an error box: its Title, Border, Normal text, and Highlighted text.

You can now select one of the components from the pop-up menu. Type the appropriate highlighted letter, and you're treated to a color palette for the item you chose. Using the arrow keys, select a color to your liking from the palette. Watch the view port to see how the item looks in that color. Press *Enter* to record your selection.

Repeat this procedure for every screen item you want to color-customize. When you are finished, press *Esc* until you are back at the main installation menu.

Note: Turbo C maintains three internal color tables: one each for color, black and white, and monochrome. TCINST allows you to change only one of these three sets of colors at a time, based upon your current video mode. For example, if you want to change to the black and white color table, you must set your video mode to BW80 at the DOS prompt and then run TCINST.

The Resize Windows Option

This option allows you to change the respective sizes of Turbo C's Edit and Message windows. Press *R* to choose **R**esize windows from the main installation menu.

Using the *Up* and *Down* arrow keys, you can move the bar dividing the Edit window from the Message window. Neither window can be smaller than three lines. When you have resized the windows to your liking, press *Enter*. You can discard your changes and return to the Installation menu by pressing *Esc*.

Quitting the Program

Once you have made all desired changes, select **Q**uit/save at the main installation menu. The message

```
Save changes to TC.EXE? (Y/N)
```

appears at the bottom of the screen.

- If you press *Y* (for Yes), all the changes you have made are permanently installed into Turbo C. (You can always run TCINST again if you want to change them.)
- If you press *N* (for No), your changes are ignored and you are returned to the operating system prompt without Turbo C's defaults or startup appearance being changed.

If you decide you want to restore the original Turbo C factory defaults, simply copy TC.EXE from your master disk onto your work disk. You can also restore the Editor commands by selecting the *E* option at the main menu, then press *R* (for Restore factory defaults) and *Esc*.

A P P E N D I X

B

TLIB: The Turbo Librarian

In this appendix, we describe TLIB, the Turbo Librarian included with Turbo C version 1.5.

What Is TLIB?

TLIB is Borland's Turbo Librarian: It is a utility that manages libraries of individual .OBJ (object module) files. A library is a very convenient way of dealing with a collection of object modules as a single unit.

The libraries included with Turbo C were built with TLIB. Using TLIB, you can build your own libraries, or you can modify the Turbo C libraries, your own libraries, libraries furnished by other programmers, or commercial libraries you have purchased. You can use TLIB to

- *create* a new library from a group of object modules
- *add* object modules or other libraries to an existing library
- *remove* object modules from an existing library
- *replace* object modules from an existing library
- *extract* object modules from an existing library
- *list* the contents of a new or existing library

When modifying an existing library, TLIB always creates a copy of the original library with a .BAK extension.

Although TLIB is not essential to creating executable programs with Turbo C, it is a useful programmer productivity tool. You will find TLIB indispensable for large development projects. If you work with object

module libraries developed by others, you can use TLIB to maintain those libraries when necessary.

The Advantages of Using Object Module Libraries

When you program in C, you often create a collection of useful C functions, like the functions in the C runtime library. Because of C's modularity, you are likely to split those functions into many separately compiled source files. You use only a subset of functions from the entire collection in any particular program. It can become quite tedious, however, to figure out exactly which files you are using. If you always include all the source files, on the other hand, your program becomes extremely large and unwieldy.

An object module library solves the problem of managing a collection of C functions. When you link your program with a library, the linker scans the library and automatically selects only those modules needed for the current program. In addition, a library consumes less disk space than a collection of object module files, especially if each of the object files is small. A library also speeds up the action of the linker, because it only opens a single file, instead of one file for each object module.

The Components of a TLIB Command Line

You run TLIB by typing a TLIB command line at the DOS prompt. To get a summary of TLIB's usage, just type `TLIB` *Enter.*

The TLIB command line takes the following general form, where items listed in square brackets (`[like this]`) are optional:

```
tlib libname [/C] [operations] [, listfile]
```

This section summarizes each of these command-line components; the following sections provide details about using TLIB. For examples of how to use TLIB, refer to the "Examples" section at the end of this appendix.

Component	*Description*
tlib	The command name that invokes TLIB.
libname	The DOS path name of the library you want to create or manage. Every TLIB command must be given a *libname.* Wildcards are not allowed. TLIB assumes an extension of .LIB if none is given. We recommend that you do not use an extension other than .LIB, since both TCC and TC's project-make facility require the .LIB extension in order to recognize library files. Note that if the named library does not exist and there are *add* operations, TLIB creates the library.
/C	The 'Case sensitive' flag. This option is not normally used; see "Advanced Operation: The /C Option " for a detailed explanation.
operations	The list of operations TLIB performs. Operations may appear in any order. If you only want to examine the contents of the library, you don't have to give any operations at all.
listfile	The name of the file listing library contents. The *listfile* name (if given) must be preceded by a comma. If you do not give a file name, no listing is produced. The listing is an alphabetical list of each module, followed by an alphabetical list of each public symbol defined in that module. You may direct the listing to the screen by using the *listfile* name CON, or to the printer by using the name PRN.

The Operation List

The operation list describes what actions you want TLIB to do. It consists of a sequence of operations given one after the other. Each operation consists of a one- or two-character *action symbol* followed by a file or module name. White space may be used around either the action symbol or the file or module name, but it cannot appear in the middle of a two-character action or in a name.

You can put as many operations as you like on the command line, up to the DOS-imposed line-length limit of 127 characters. The order of the operations is not important. TLIB always applies the operations in a specific order:

1. All extract operations are done first.
2. All remove operations are done next.
3. All add operations are done last.

Replacing a module is treated as first removing it, then adding the replacement module.

File and Module Names

When TLIB adds an object module file to a library, the file is simply called a *module*. TLIB finds the name of a module by taking the given file name and stripping any drive, path, and extension information from it. (Typically, drive, path, and extension are not given.)

Note that TLIB always assumes reasonable defaults. For example, to add a module that has an .OBJ extension from the current directory, you only need to supply the module name, not the path and .OBJ extension.

Wildcards are never allowed in file or module names.

TLIB Operations

TLIB recognizes three action symbols (`-`, `+`, `*`), which you can use singly or combined in pairs for a total of five distinct operations. For operations that use a pair of characters, the order of the characters in not important. The action symbols and what they do are listed here:

Action Symbol	*Name*	*Description*
+	Add	TLIB adds the named file to the library. If the file has no extension given, TLIB assumes an extension of .OBJ. If the file is itself a library (with a .LIB extension), then the operation adds all of the modules in the named library to the target library.

		If a module being added already exists, TLIB displays a message and does not add the new module.
-	**Remove**	TLIB removes the named module from the library. If the module does not exist in the library, TLIB displays a message.
*	**Extract**	TLIB creates the named file by copying the corresponding module from the library to the file. If the module does not exist, TLIB displays a message and does not create a file. If the named file already exists, it is overwritten.
-+ +-	**Replace**	TLIB replaces the named module with the corresponding file. This is just a shorthand for a *remove* followed by an *add* operation.
-* *-	**Extract & Remove**	TLIB copies the named module to the corresponding file name and then removes it from the library. This is just a shorthand for an *extract* followed by a *remove* operation.

A remove operation only needs a module name, but TLIB allows you to enter a full path name with drive and extension included. However, everything but the module name is ignored.

It is not possible to rename modules in a library. To rename a module, you first must extract and remove it, rename the file just created, and, finally, add it back into the library.

Creating a Library

To create a library, you simply add modules to a library that does not yet exist.

Using Response Files

When you are dealing with a large number of operations, or if you find yourself repeating certain sets of operations over and over, you will probably want to start using *response files*. A response file is simply an ASCII text file (easily created with Turbo C editor) that contains all or part of a TLIB command. Using response files, you can build TLIB commands larger than would fit on one DOS command line.

To use a response file *pathname*, specify `@<pathname>` at any position on the TLIB command line.

- More than one line of text can make up a response file; you use the "and" character (`&`) at the end of a line to indicate that another line follows.
- You don't need to put the entire TLIB command in the response file; the file can provide a portion of the TLIB command line, and you can type in the rest.
- You can use more than one response file in a single TLIB command line.

See "Examples" for a sample response file and a TLIB command line incorporating it.

Advanced Operation: The /C Option

When you add a module to a library, TLIB maintains a dictionary of all public symbols defined in the modules of the library. All symbols in the library must be distinct. If you try to add a module to the library that would cause a duplicate symbol, TLIB will display a message and not add the module.

Normally, when TLIB checks for duplicate symbols in the library, uppercase and lowercase letters are not considered as distinct. For example, the symbols *lookup* and *LOOKUP* are treated as duplicates. Since C *does* treat uppercase and lowercase letters as distinct, you need to use the `/c` option to add a module to a library that includes a symbol that differs *only in case* from one already in the library. The `/c` option forces TLIB to accept a module with symbols in it that differ only in case from symbols already in the library.

It may seem odd that, without the `/c` option, TLIB rejects symbols that differ only in case, especially since C is a case-sensitive language. The reason is that some linkers fail to distinguish between symbols in a library that differ only in case.

TLINK has no problem distinguishing uppercase and lowercase symbols, and it will properly accept a library containing symbols that differ only in case. As long as you only use the library with TLINK, you can use the TLIB `/c` option without any problems.

However, if you want to use the library with other linkers (or allow other people to use the library with other linkers), for your own protection you should not use the `/c` option.

Examples

Here are some simple examples demonstrating the different things you can do with TLIB.

1) To create a library named MYLIB.LIB with modules X.OBJ, Y.OBJ, and Z.OBJ, type

```
tlib mylib +x +y +z
```

2) To create a library as in #1 and get a listing, too, type

```
tlib mylib +x +y +z, mylib.lst
```

3) To get a listing of an existing library CS.LIB, type

```
tlib cs, cs.lst
```

4) To replace module X.OBJ with a new copy, add A.OBJ and delete Z.OBJ from MYLIB.LIB, type

```
tlib mylib -+x +a -z
```

5) To extract module Y.OBJ from MYLIB.LIB and get a listing, type

```
tlib mylib *y, mylib.lst
```

6) To create a new library with modules A.OBJ, B.OBJ, ..., G.OBJ using a response file:

   First create a text file, ALPHA.RSP, with

```
+a.obj +b.obj +c.obj &
+d.obj +e.obj +f.obj &
+g.obj
```

   Then use the TLIB command:

```
tlib alpha @alpha.rsp, alpha.lst
```

A P P E N D I X

C

GREP: A File-Search Utility

In this appendix, we describe GREP.COM, Turbo C's very fast version of the well-known UNIX file-search utility.

What Is GREP?

GREP is a powerful search utility that can search for text in several files at once.

The general command-line syntax for GREP is:

```
grep [options] searchstring filespec [filespec filespec ... filespec]
```

For example, if you want to see in which source files you call the **setupmodem** function, you could use GREP to search the contents of all the .C files in your directory to look for the string **setupmymodem**, like this:

```
grep setupmodem *.c
```

The GREP Options

In the command line, *options* are one or more single characters preceded by a dash symbol (-). Each individual character is a switch that you can turn *on* or *off*: type the plus symbol (+) after a character to turn the option *on*, or type a dash (-) after the character to turn the option *off*.

The default is *on* (the + is implied): for example, `-r` means the same thing as `-r+`. You can list multiple options individually (like this: `-i -d -l`) or you

can combine them (like this: `-ild` or `-il -d`, etc.): they're all the same to GREP.

Here is a list of the option characters used with GREP and their meanings:

–c *Count only*: Only a count of matching lines is printed. For each file that contains at least one matching line, GREP prints the file name and a count of the number of matching lines. Matching lines are not printed.

–d *Directories*: For each *filespec* specified on the command line, GREP searches for all files that match the file specification, both in the directory specified *and* in all subdirectories below the specified directory. If you give a *filespec* without a path, GREP assumes the files are in the current directory.

–i *Ignore case*: GREP ignores upper/lowercase differences (case folding). GREP treats all letters *a-z* as being identical to the corresponding letters *A-Z* in all situations.

–l *List match files*: Only the name of each file containing a match is printed. After GREP finds a match, it prints the file name and processing immediately moves on to the next file.

–n *Numbers*: Each matching line that GREP prints is preceded by its line number.

–r *Regular expression search*: The text defined by *searchstring* is treated as a regular expression instead of as a literal string.

–v *Non-match*: Only non-matching lines are printed. Only lines that *do not* contain the search string are considered to be non-matching lines.

–w *Write options*: GREP will combine the options given on the command line with its default options and write these to the GREP.COM file as the new defaults. (In other words, GREP is self-configuring.) This option allows you to tailor the default option settings to your own taste.

–z *Verbose*: GREP prints the file name of every file searched. Each matching line is preceded by its line number. A count of matching lines in each file is given, even if the count is zero.

Order of Precedence

A few of GREP's options override certain others; the following order of precedence applies:
`-z` overrides `-l -c -n`
`-l` overrides `-c -n`
`-c` overrides `-n`

For example, suppose you type in the following GREP command line:

```
grep -c -z main( my*.c
```

GREP will search all files matching the MY*.C file specification for the search string *main(* and print the file name of every file searched, number each matching line, *and* give a count of matching lines for each file searched.

Remember that each option is a switch: its state reflects the way you last "flipped" it. At any given time, each option can only be *on* or *off*. Each occurrence of a given option on the command line overrides its previous definition. For example, you might type in the following command line:

```
grep -r -i- -d -i -r-  main( my*.c
```

Given this command line, GREP will run with the `-d` option *on*, the `-i` option *on*, and the `-r` option *off*.

You can install your preferred default setting for each option in GREP.COM with the `-w` option. For example, if you want GREP to always do a verbose search (`-z` *on*), you can install it with the following command:

```
grep -w -z
```

The Search String

The value of *searchstring* defines the pattern GREP will search for. A search string can be either a *regular expression* or a *literal string*. In a regular expression, certain characters have special meanings: they are operators that govern the search. In a literal string, there are no operators: each character is treated literally.

You can enclose the search string in quotation marks to prevent spaces and tabs from being treated as delimiters. Matches will not cross line boundaries (a match must be contained in a single line).

An expression is either a single character or a set of characters enclosed in brackets. A concatenation of regular expressions is a regular expression.

Operators in Regular Expressions

When you use the `-r` option, the search string is treated as a regular expression (not a literal expression) and the following characters take on special meanings:

^ A circumflex at the start of the expression matches the start of a line.

$ A dollar sign at the end of the expression matches the end of a line.

. A period matches any character.

* An expression followed by an asterisk wildcard matches zero or more occurrences of that expression. For example: in `fo*`, the `*` operates on the expression *o*; it matches *f, fo, foo,* etc. (*f* followed by zero or more *o*s), but doesn't match *fa.*

+ An expression followed by a plus sign matches one or more occurrences of that expression: `fo+` matches *fo, foo,* etc., but not *f.*

[ ] A string enclosed in brackets matches any character in that string, but no others. If the first character in the string is a circumflex (^), the expression matches any character *except* the characters in the string. For example, `[xyz]` matches *x, y,* or *z,* while `[^xyz]` matches *a* and *b,* but not *x, y,* or z. You can specify a range of characters with two characters separated by a dash (-). These can be combined to form expressions (like `[a-bd-z?]` to match ? and any lowercase letter except *c*).

\ The *backslash escape character* tells GREP to seach for the literal character that follows it. For example, `\.` matches a period instead of "any character".

Note: Four of the previously-described characters (`$` `.` `*` and `+`) do not have any special meaning when used within a bracketed set. In addition, the character `^` is only treated specially if it immediately follows the beginning of the set definition (that is, immediately after the `[`).

Any ordinary character not mentioned in the preceding list matches that character. (> matches >, `#` matches `#`, etc.)

The File Specification

The third item in the GREP command line is *filespec,* the file specification; it tells GREP which files (or groups of files) to search. *filespec* can be an

explicit file name, or a "generic" file name incorporating the DOS `?` and `*` wildcards. In addition, you can enter a path (drive and directory information) as part of *filespec*. If you give *filespec* without a path, GREP only searches the current directory.

Examples with Notes

The following examples assume that all of GREP's options default to *off*:

Example 1

Command line: `grep main( *.c`

Matches:

```
main()
mymain(
```

Does not match:

```
mymainfunc()
MAIN(i: integer);
```

Files Searched: *.C in current directory.

Note: By default, the search is case-sensitive.

Example 2

Command line: `grep -r [^a-z]main\ *( *.c`

Matches:

```
main(i:integer)
main(i,j:integer)
if (main  ()) halt;
```

Does not match:

```
mymain()
MAIN(i:integer);
```

Files Searched: *.C in current directory.

Note: The search string here tells GREP to search for the word *main* with no preceding lowercase letters (`[^a-z]`), followed by zero or more occurrences of blank spaces (`\ *`), then a left parenthesis.

Since spaces and tabs are normally considered to be command-line delimiters, you must *quote* them if you want to include them as part of a regular expression. In this case, the space after *main* is quoted with the backslash escape character. You could also accomplish this by placing the space in double quotes (`[^a-z]main" "*`).

Example 3

Command line: `grep -ri [a-c]:\\data\.fil *.c *.inc`

Matches:
```
A:\data.fil
c:\Data.Fil
B:\DATA.FIL
```

Does not match:
```
d:\data.fil
a:data.fil
```

Files Searched: *.C and *.INC in current directory.

Note: Because the backslash and period characters (\ and .) usually have special meaning, if you want to search for them, you must quote them by placing the backslash escape character immediately in front of them.

Example 4

Command line: `grep -ri [^a-z]word[^a-z] *.doc`

Matches:
```
every new word must be on a new line.
MY WORD!
word--smallest unit of speech.
In the beginning there was the WORD, and the WORD
```

Does not match:
```
Each file has at least 2000 words.
He misspells toward as toword.
```

Files Searched: *.DOC in the current directory.

Note: This format basically defines how to search for a given word.

Example 5

Command line: `grep "search string with spaces" *.doc *.asm a:\work\myfile.*`

Matches:
```
This is a search string with spaces in it.
```

Does not match:
```
THIS IS A SEARCH STRING WITH SPACES IN IT.
This is a search string with many spaces in it.
```

Files Searched: *.DOC and *.ASM in the current directory, and MYFILE.* in a directory called \WORK on drive A:.

Note: This is an example of how to search for a string with embedded spaces.

Example 6

Command line: `grep -rd "[ ,.:?'\"]"$ \*.doc`

Matches:

```
He said hi to me.
Where are you going?
Happening in anticipation of a unique situation,
Examples include the following:
"Many men smoke, but fu man chu."
```

Does not match:

```
He said "Hi" to me
Where are you going? I'm headed to the beach this
```

Files Searched: *.DOC in the root directory and all its subdirectories on the current drive.

Note: This example searches for the characters `,.:?'` and `"` at the end of a line. Notice that the double quote within the range is preceded by an escape character so it is treated as a normal character instead of as the ending quote for the string. Also, notice how the `$` character appears outside of the quoted string. This demonstrates how regular expressions can be concatenated to form a longer expression.

Example 7

Command line: `grep -ild " the " \*.doc`
or `grep -i -l -d " the " \*.doc`
or `grep -il -d " the " \*.doc`

Matches:

```
Anyway, this is the time we have
do you think? The main reason we are
```

Does not match:

```
He said "Hi" to me just when I
Where are you going? I'll bet you're headed to
```

Files Searched: *.DOC in the root directory and all its subdirectories on the current drive.

Note: This example ignores case and just prints the names of any files that contain at least one match. The three examples show different ways of specifying multiple options.

A P P E N D I X

D

BGIOBJ: Conversion Utility for Graphics Drivers and Fonts

In this appendix, we explain how to use BGIOBJ, a utility that allows you to use a non-dynamic scheme for loading graphics drivers and character fonts into your graphics programs.

What Is BGIOBJ?

BGIOBJ is a utility you can use to convert graphics driver files and character sets (stroked font files) to object files. Once they're converted, you can link them into your program, making them part of the executable file. This is offered in addition to the graphics package's dynamic loading scheme, in which your program loads graphics drivers and character sets (stroked fonts) from disk at execution time.

Linking drivers and fonts directly into your program is advantageous because the executable file contains all (or most) of the drivers and/or fonts it might need, and doesn't need to access the driver and font files on disk when running. However, linking the drivers and fonts into your executable increases its size.

To convert a driver or font file to a linkable object file, use the BGIOBJ.EXE utility. This is the simplified syntax:

```
BGIOBJ <source file>
```

where *<source file>* is the driver or font file to be converted to an object file. The object file created has the same file name as the source file, with the

extension .OBJ; for example, EGAVGA.BGI yields EGAVGA.OBJ, SANS.CHR gives SANS.OBJ, etc.

Adding the New .OBJ Files to GRAPHICS.LIB

You should add the driver and font object modules to GRAPHICS.LIB, so the linker can locate them when it links in the graphics routines. If you don't add these new object modules to GRAPHICS.LIB, you'll have to add them to the list of files in the TC project (.PRJ) file, on the TCC command line, or on the TLINK command line. To add these object modules to GRAPHICS.LIB, invoke the Turbo Librarian (TLIB) with the following command line:

```
tlib graphics +<object file name> [ +<object file name> ... ]
```

where *<object file name>* is the name of the object file created by BGIOBJ.EXE (such as CGA, EGAVGA, GOTH, etc.); the .OBJ extension is implied, so you don't need to include it. You can add several files with one command line to save time; see the example following.

(For more information about TLIB, refer to Appendix B in this addendum.)

Registering the Drivers and Fonts

After adding the driver and font object modules to GRAPHICS.LIB, you have to *register* all the drivers and fonts that you want linked in; you do this by calling **registerbgidriver** and **registerbgifont** in your program (before calling **initgraph**). This informs the graphics system of the presence of those files, and ensures that they will be linked in when the executable file is created by the linker.

The registering routines each take one parameter; a symbolic name defined in GRAPHICS.H. Each registering routine returns a non-negative value if the driver or font is successfully registered.

The following table is a complete list of drivers and fonts included with Turbo C. It shows the names to be used with **registerbgidriver** and **registerbgifont**.

Driver file (*.BGI)	**registerbgidriver** *Symbolic name*	*Font file* (*.CHR)	**registerbgifont** *Symbolic name*
CGA	CGA_driver	TRIP	triplex_font
EGAVGA	EGAVGA_driver	LITT	small_font
HERC	Herc_driver	SANS	sansserif_font
ATT	ATT_driver	GOTH	gothic_font
PC3270	PC3270_driver		

An Example

Here's a complete example. Suppose you want to convert the files for the CGA graphics driver, the gothic font, and the triplex font to object modules, then link them into your program .

1. Convert the binary files to object files using BGIOBJ.EXE, as shown in the following separate command lines:

```
bgiobj  cga
bgiobj  trip
bgiobj  goth
```

   This creates 3 files: CGA.OBJ, TRIP.OBJ, and GOTH.OBJ.

2. You can add these object files to GRAPHICS.LIB with this TLIB command line:

```
tlib  graphics  +cga  +trip  +goth
```

   If you don't add the object files to GRAPHICS.LIB, you need to add the object file names CGA.OBJ, TRIP.OBJ, and GOTH.OBJ to your project list (if you are using Turbo C's integrated environment), or to the TCC command line. For example, the TCC command line would look like this:

```
tcc  niftgraf graphics.lib  cga.obj  trip.obj  goth.obj
```

3. You register these files in your graphics program like this:

```
/* header file declares CGA_driver, triplex_font, and gothic_font */
#include <graphics.h>

/* register and check for errors (one never knows ....) */

if (registerbgidriver(CGA_driver) < 0) exit(1);
if (registerbgifont(triplex_font) < 0) exit(1);
if (registerbgifont(gothic_font)  < 0) exit(1);
```

```
/* ... */
initgraph(....);          /* initgraph should be called after registering */
/* ... */
```

If you ever get a linker error `Segment exceeds 64k` after linking in some drivers and/or fonts, refer to the following section.

The /F option

This section explains what steps to take if you get the linker error `Segment exceeds 64k` (or a similar error) after linking in several driver and/or font files (especially with tiny, small, and compact model programs).

By default, the files created by BGIOBJ.EXE all use the same segment (called _TEXT). This can cause problems if your program links in many drivers and/or fonts, or when you're using the tiny, small, or compact memory model.

There is NO cure if this happens in tiny model programs. You will have to unlink some or all of the drivers and fonts, and use the dynamic driver/font loading scheme.

With other model programs, you can convert one or more of the drivers or fonts with the BGIOBJ `/F` option. This option directs BGIOBJ to use a segment name of the form <*filename*>_TEXT, so that the default segment is not overburdened by all the linked-in drivers and fonts (and, in small and compact model programs, all the program code). For example, the following two BGIOBJ command lines direct BGIOBJ to use segment names of the form EGAVGA_TEXT and SANS_TEXT.

```
bgiobj  /F  egavga
bgiobj  /F  sans
```

When you select the `/F` option, BGIOBJ also appends F to the target object file (EGAVGAF.OBJ, SANSF.OBJ, etc.), and appends *_far* to the name that will be used with **registerfarbgidriver** and **registerfarbgifont**. (For example, *EGAVGA_driver* becomes *EGAVGA_driver_far*.) For files created with `/F`, you must use these far registering routines instead of the regular **registerbgidriver** and **registerbgifont**. For example:

```
if (registerfarbgidriver(EGAVGA_driver_far) < 0) exit(1);
if (registerfarbgifont(sansserif_font_far)  < 0) exit(1);
```

Advanced BGIOBJ Features

This section explains some of BGIOBJ's advanced features, and the routines **registerfarbgidriver** and **registerfarbgifont**. Only experienced users should use these features.

This is the full syntax of the BGIOBJ.EXE utility:

```
BGIOBJ  [/F] <source> <destination> <public name> <seg-name> <seg-class>
```

Component	*Description*
`/F` or `-F`	This option instructs BGIOBJ.EXE to use a segment name other than _TEXT (the default), and to change the public name and destination file name. (See the previous section for a detailed discussion of `/F`.)
`<source>`	This is the driver or font file to be converted. If the file is not one of the driver/font files shipped with Turbo C, you should specify a full file name (including extension).
`<destination>`	This is the name of the object file to be produced. The default destination file name is *<source>*.OBJ, or *<source>*F.OBJ if you use the `/F` option.
`<public name>`	This is the name that will be used in the program in a call to **registerbgidriver** or **registerbgifont** (or their respective **far** versions) to link in the object module. The public name is the external name used by the linker, so it should be the name used in the program, prefixed with an underbar. If your program uses Pascal calling conventions, use only uppercase letters, and do not add an underbar.
`<seg-name>`	This is an optional segment name; the default is _TEXT (or *<filename>*_TEXT if `/F` is specified)
`<seg-class>`	This is an optional segment class; the default is CODE.

All parameters except *<source>* are optional. If you need to specify an optional parameter, all the parameters preceding it must also be specified.

If you choose to use your own public name(s), you have to add declaration(s) to your program, using one of the following forms:

```
void public_name(void);          /* if /F not used, default segment name used */

extern int far public_name[];       /* if /F used, or segment name not _TEXT */
```

In these declarations, `public_name` matches the *<public name>* you used when converting with BGIOBJ. The GRAPHICS.H header file contains declarations of the default driver and font public names; if you use those default public names you don't have to declare them as just described.

After these declarations, you have to register all the drivers and fonts in your program. If you don't use the `/F` option and don't change the default segment name, you should register through **registerbgidriver** and **registerbgifont**; otherwise use **registerfarbgidriver** and **registerfarbgifont**.

Index

A

B

C

D

E

H

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

BORLAND